JOÃO JOSÉ NETO

"ASPECTOS DO PROJETO DE SOFTWARE DE UM MINICOMPUTADOR"

"Dissertação de Mestrado" apresentada à Escola Politécnica da Universidade de São Paulo, para obtenção do título de Mestre em Engenharia.

Área de Concentração - Engenharia de Eletricidade.

Orientador: Prof. Dr. Antonio Marcos de Aguirra Massola

São Paulo
-1975-

Ao Prof. [illegible] ofereço um exemplar do meu trabalho de mestrado, com meus agradecimentos. 10/11/75

JOÃO JOSÉ NETO

"ASPECTOS DO PROJETO DE SOFTWARE DE UM MINICOMPUTADOR"

"Dissertação de Mestrado" apresentada à Escola Politécnica da Universidade de São Paulo, para obtenção do título de Mestre em Engenharia.

Área de Concentração - Engenharia de Eletricidade.

Orientador: Prof. Dr. Antonio Marcos de Aguirra Massola

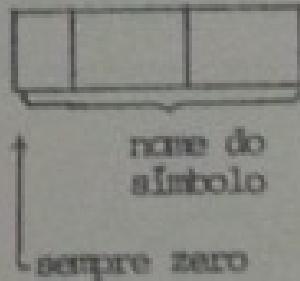
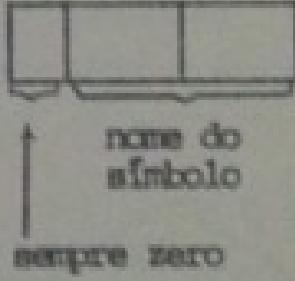
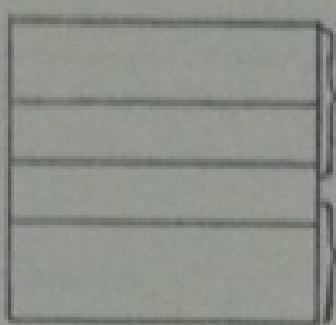
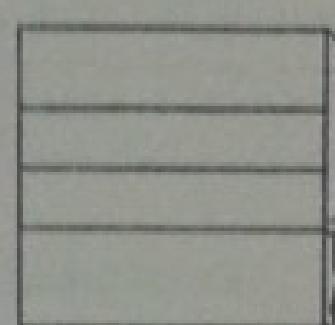
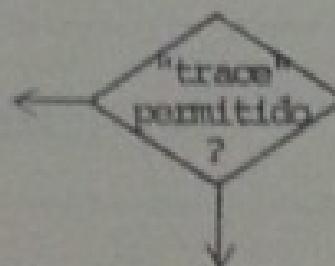
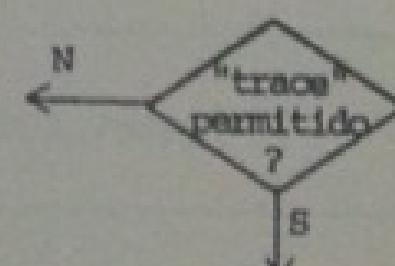
São Paulo
-1975-

À meus pais e irmã.

A G R A D E C I M E N T O S

Ao Prof. Dr. Antonio Marcos de Aguirra Massola, pela dedicação com que acompanhou, orientou, apoiou e incentivou este trabalho.

Aos Professores Doutores James Gregory Rudolph, Antonio Hélio Guerra Vieira e Tamio Shimizu pelas idéias apresentadas.

Aos Engs. Benício José de Souza, Ting Kong Sen e Wanner Monteiro Pinheiro, e engenheirandos Luiz Sanches Filho, Mário Tachibana e Charlie Lin, pela participação efetiva no projeto e no aprimoramento dos programas desenvolvidos.

Aos Engs. Laércio Antonio Marzagão, Antonio Marcos de Aguirra Massola e Benício José de Souza, à Enga. Selma Shin Shimizu Melnikoff, à Profa. Selenê Cavalcanti Ferrari e aos meus familiares, pelos inúmeros diálogos e constante apoio e estímulo, decisivos para a concretização deste trabalho.

À srta. Sonia Regina Izarelli pelos serviços de datilografia e desenho.

A Valdecir Finco pelo serviços de impressão.

A todos, enfim, que de uma ou outra forma contribuíram para a concepção ou desenvolvimento deste trabalho, particularmente à equipe de estagiários do Laboratório de Sistemas Digitais, sem cuja valiosa participação não teria sido possível a realização deste trabalho.

"E R R A T A"

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
1	5	Digitiais	Digitais
5	28	pseudos instruções	pseudo instruções
7	3	filosofia de rotina	filosofia da rotina
	7	de linguagem	da linguagem
21	14	do teclado	no teclado
	27	"linedeed"	"linefeed"
22	28	o <u>instruções de</u>	• <u>instruções de</u>
33	19	indicado	definido
	26	ref. 7	ref. 4
38	12	quatro zeros	quatro nulos
39	11	e assim subtrair	e assim por diante, bastará subtrair
43	Fig. 2.5.2.2	nome do símbolo; sempre zero	nome do símbolo; sempre zero
44	1 e 2	... "cross refe-rence table" ...	... "cross refer-ence table" ...
	25	"hasj"	"hash"
	29	Constatando	Constando

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
46	Fig. 2.5.3.1		
49	9	do pseudo FIM	da pseudo FIM
63	21	pode-se-ia	poder-se-ia
65	última	ponteiro para a	endereço da
69	1	definido	definindo
	11	o "menos significativos"	o menos significativo
	21	pode assumir	pode admitir
	29	pela comando BLT	pelo comando BLT
73	penúltima	gerado no passo 1	gerada no passo 1
81	9	capacidade de tabela	capacidade da tabela
89	1	Representação de um interface	Representação de uma interface
94		"trace" permitido ?	N "trace" permitido ? S
96	17	Liga todos os "flipflops"	Limpa todos os "flipflops"

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
98	2	da fase de Console	na fase de Console
	7	proxima instrução	próxima instrução
103	ante-penúl-tima	convivência	conveniência
115	ante-penúl-tima	durante o objeto	durante o projeto
127	24	a mesma lógica	lógicas diferentes
134	18	em 3.6).	em 3.5).
	27	sugerido; a causa	sugerido, a causa
139	25	é possível, implemen tar	é possível implemen tar
145	último bloco de (3)	perfurar linha lida Console	perfurar linha lida na Console
146	última	seja o fim-de-fita	seja o de fim-de-fita
147	4	necessário a	necessário à
	16	da saída	de saída
156	Fig. A1.3	DESLI/GIRO	DESL./GIRO
162	1a. li-nha da tabela A1.6.1	Limpe Acumulador e V	Limpa Acumulador e V

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
163	1	Possui	Possuem
165	21	endereços externos	endereços extremos
168	12	"bytes"de região D	"bytes" da região D
	15	"bytes"de região D	"bytes" da região D
170	2	(ref. 9)	(ref. 10)
171	Fig. A3.2.2	A	Sequência de N "bytes"
172	Fig. A3.2.3	comprimento de área comum	comprimento da área comum
173	Fig. A3.2.4	$E_{2^{n-1}}$ E_{2^n} $R_{2^{n-1}}$ R_{2^n}	E_{2n-1} E_{2n} R_{2n-1} R_{2n}
176	Fig. A3.2.7	os operandos relocá veis e	os operandos relocá-veis e "comuns"
183	ante-penúl-tima	estidado	editado
190	15	Digitiais	Digitais
191	3	Publishing Co,	Publishing Co, 1969
	9	John Wiley Sons	John Wiley & Sons

R E S U M O

O presente trabalho descreve os métodos utilizados no desenvolvimento de alguns dos módulos do "software" básico do Patinho Feio, o primeiro minicomputador desenvolvido no Laboratório de Sistemas Digitiais do Departamento de Engenharia de Eletricidade da Escola Politécnica da Universidade de São Paulo.

Para cada módulo apresentado, são discutidos os seus objetivos, sendo, quando conveniente, apresentados alguns dos problemas enfrentados durante seu desenvolvimento específico ou então problemas mais gerais, referentes ao projeto de programas semelhantes ao mesmo. Sempre que possível, são apresentadas alternativas de solução dos referidos problemas, descrevendo-se finalmente alguns detalhes de um exemplo de implementação.

No Capítulo 2 são discutidos mais profundamente os problemas enfrentados durante o projeto e a implementação de um montador, desde a sua concepção até a sua implantação definitiva, enfatizando-se a metodologia empregada na implementação e os critérios adotados nas decisões mais importantes.

No Capítulo 3 é descrito um simulador-interpretador, implementado em outro computador com a intenção de auxiliar o desenvolvimento do "software" básico do Patinho Feio. Algumas técnicas de simulação são discutidas neste Capítulo, ao lado dos algoritmos utilizados pelo interpretador.

Nos demais Capítulos, são descritos mais alguns programas do "software" básico desenvolvidos para o Patinho Feio, tais como um desmontador, um programa para auxiliar a depuração de outros programas e um editor simbólico.

Nos apêndices, são apresentados tópicos julgados convenientes para a complementação de algumas idéias, bem como alguns exemplos de utilização de programas apresentados neste trabalho.

A B S T R A C T

The present work describes the methods that had been employed during the development of some of the basic software modules of the "Patinho Feio", the first minicomputer of the "Laboratório de Sistemas Digitais da Escola Politécnica da Universidade de São Paulo".

For each module, after a brief discussion of their objectives, some of the main problems which had to be solved during their development are presented, as well as more general ones, which arrive when designing programs of the same class. Whenever possible, alternative solutions of these problems are presented, and, at last, an example of implementation is described.

Chapter 2 discusses in detail the problems arrived during the design and implementation of an assembler, from the phase of its conception until that of its final installation, emphasizing the methods employed in the implementation and the criteria which were used when the most important decisions had to be taken.

Chapter 3 describes a simulator-interpreter, which had been implemented in an auxiliary computer, and was intended to help the development of the basic software of the "Patinho Feio".This chapter discusses also some simulation techniques and the algorithms employed in the interpreter.

The remaining chapters describe some additional programs of the basic software developed for the "Patinho Feio", like a disassembler, a debugging routine and a symbolic editor.

The appendices present some subjects considered helpful for completing some ideas, and some execution examples of the programs presented in this work.

Í N D I C E

CAPÍTULO 1. INTRODUÇÃO

1. INTRODUÇÃO

1.1 - Objetivos

No presente trabalho expõe-se os métodos utilizados no projeto e construção de algumas das peças do "software" básico do Patinho Feio, o primeiro minicomputador desenvolvido no Laboratório de Sistemas Digitais (LSD) do Departamento de Engenharia de Eletricidade da Escola Politécnica da Universidade de São Paulo. Descreve-se as diversas fases do projeto, analisando detalhadamente os principais problemas que cada uma delas apresenta. Esta análise é feita em nível geral, quando se tratar de problemas que devam ser enfrentados independentemente da máquina, ou então em nível particular,em caso contrário. Esta descrição destina-se principalmente à orientação de futuros projetos semelhantes.

A análise dos problemas dependentes da máquina, bem como a dos detalhes de implementação dos programas descritos, restringe-se ao caso particular dos exemplos implementados, devendo-se levar em conta a configuração do sistema para o qual foram desenvolvidos. Assim, algumas das soluções apresentadas em tais exemplos podem não ser vantajosas em sistemas que não tenham as mesmas características.

São apresentadas alternativas para as soluções de alguns dos problemas enfrentados, fazendo-se uma análise comparativa das mesmas. Procurou-se, para isto, ressaltar vantagens e desvantagens de cada uma das alternativas mencionadas, terminado-se por optar por uma delas e, sempre que possível, justificando-se a opção adotada na implementação do exemplo, com uma argumentação baseada nas características e limitações do computador utilizado.

Além disto, é descrito aqui o funcionamento dos programas implementados, especialmente com a finalidade de documentar as linhas de raciocínio e a filosofia de projeto e de implementação adotadas para facilitar futuras alterações de tais programas.

1.2 - Generalidades (ref. 11, 14)

Nos próximos Capítulos estão desenvolvidos, dentro das linhas apresentadas em 1.1, os seguintes módulos do "software" :

- um montador ("assembler"), cuja finalidade é a de dispensar o programador da tarefa de programação em linguagem de máquina, permitindo que os programas sejam escritos em linguagem simbólica mnemônica. Uma das versões implementadas dá, além disto, a possibilidade de modularização dos programas, introduzindo a facilidade de se utilizar programas escritos e traduzidos independentemente sem a necessidade de nova tradução a cada utilização dos programas parciais;

- uma rotina de depuração, cuja finalidade é a de facilitar a pesquisa e correção de erros existentes em programas ainda não depurados;

- um desmontador ("disassembler"), cuja finalidade é a de gerar programas escritos na linguagem do montador, a partir de uma fita binária contendo o programa correspondente em linguagem de máquina;

- uma rotina de edição de arquivos ASCII, cuja finalidade é a de mecanizar a correção, a reprodução, a modificação e a listagem de arquivos ASCII, como por exemplo, programas perfurados, em linguagem simbólica, em fita de papel ("programas fonte").

Os quatro programas citados acima foram desenvolvidos para o Patinho Feio estando atualmente à disposição para execução no mesmo.

Alguns programas de utilidade foram implementados no computador Hewlett-Packard 2116-B, com a finalidade de facilitar e de tornar mais versátil a utilização do "software" do Patinho Feio.

Escolheu-se o computador HP-2116-B principalmente pela sua compatibilidade de entrada e saída com a do Patinho Feio (fita de papel). Os programas desenvolvidos para o HP-2116-B não foram implementados diretamente no Patinho Feio devido a algumas restrições atualmente existentes no mesmo, como por exemplo sua pequena capacidade de memória, a inexistência de memória de massa e a baixa velocidade dos periféricos disponíveis. Entretanto, poderão alguns desses programas vir a ser implantados no Patinho Feio assim que estiverem disponíveis no mesmo os periféricos necessários. Os programas aqui apresentados são os seguintes:

- um simulador em nível de registradores, que funciona como um interpretador da linguagem de máquina do Patinho Feio e que também permite a utilização dos periféricos rápidos do HP-2116-B em substituição aos disponíveis no Patinho Feio, bem como o acompanhamento do programa em execução por meio de rastreamento ("traces"), descarregamentos ("dumps") de memória, monitoramento dos processos de entrada e saída, etc. Naturalmente a utilização deste simulador-interpretador é às vezes mais conveniente que a do próprio Patinho Feio, como por exemplo, quando o programa for limitado em velocidade pela entrada e saída (caso de rastreamento e de listagens extensas com pouco processamento);

- um montador ("cross-assembler") equivalente ao desenvolvido para o Patinho Feio, que incorpora uma opção de geração da tabela de referências cruzadas ("cross-reference symbol table") e ordenação alfabética da tabela de símbolos, permitindo utilização total dos recursos do sistema operacional DOS (" disc operating system") do computador HP-2116-B. A tabela de referências cruzadas tem como finalidade facilitar a documentação e mesmo a depuração de programas extensos, gerando, a partir da linguagem simbólica, uma tabela em ordem alfabética, de todos os rótulos ("labels", isto é, nomes simbólicos dados aos endereços das instruções ou dos dados do programa), ao lado de cada qual são listados o número da linha onde o mesmo foi definido e o conjunto de todas as linhas onde foi referenciado. Indica também os rótulos indefinidos, os não utilizados, e o endereço de memória associado a cada rótulo.

1.3 - Observações

- Os dois computadores do LSD em que se efetuou a elaboração dos programas descritos neste trabalho, foram utilizados - com a seguinte configuração:

PATINHO FEIO:

1 Unidade Central de Processamento, com memória de núcleos de ferrite com 4K palavras de 8 bits (1K = 1024).

1 Leitora Ótica de Fita de Papel HP-2737-A, 300 caracteres por

segundo (máximo).

2 Terminais Teletype ASR33 com leitora e perfuradora de fita de papel, entrada por teclado e saída impressa, 1Ø caracteres por segundo (perfuração opcional).

HEWLETT-PACKARD 2116-B:

1 Unidade Central de Processamento, com memória de núcleos de ferrite com 16K palavras de 16 bits.

1 Unidade de Disco Magnético HP-2770-A, com capacidade de 3 Megabits, organizados em 64 trilhas, de 92 setores cada, tendo cada setor 64 palavras de 17 bits.

1 Impressora HP-2767-A de 8Ø colunas por linha, 111Ø linhas por minuto (máximo).

3 Unidades de Fita Magnética HP-7970-A, velocidade máxima 37.5 ips, densidade 8ØØ bpi.

1 Console Teletype ASR35, de 1ØØ caracteres por minuto.

1 Leitora Ótica de Fita de Papel HP-2748-B, de 5ØØ caracteres por segundo (máximo).

1 Perfuradora de Fita de Papel HP-2895-B, de 75 caracteres por segundo.

1 Leitora de Cartões de marcas óticas HP-2761-A, de 2ØØ cartões por minuto.

- Os programas desenvolvidos para o Patinho Feio foram inicialmente projetados para a configuração mínima acima descrita. Entretanto, com a inclusão de novos periféricos, é conveniente que se configure os programas existentes, adaptando-os aos novos equipamentos disponíveis, o que pode tornar mais rápida e efi ciente a execução de tais programas. Assim, todas as vezes que um

novo periférico é incorporado ao sistema, faz-se a adaptação dos programas às novas condições, permitindo que sejam utilizados os recursos implantados.

Atualmente, dispõe-se dos seguintes periféricos adicionais, já incorporados ao sistema pela equipe do LSD:

1 Impressora de linha de 8∅ colunas HP-2767-A (111∅ linhas/minuto).

1 Terminal DECWRITER DEC LA 3∅S, de 8∅ colunas, 3∅ caracteres por segundo (máximo).

1 Perfuradora Rápida de Fita de Papel HP-2735-A, de 115 caracteres por segundo.

Os programas menos extensos permitem a inclusão de uma "seção de configuração", parte do programa que o adapta para a utilização dos periféricos desejados. Os mais extensos, no entanto, não dispõem de memória livre suficiente que possa alojar a seção de configuração. Para estes programas, a configuração pode ser feita manualmente modificando-se certas posições de memória, ou, então montando-se novamente o programa com as devidas alterações.

- Em todos os programas aqui descritos, procurou-se orientar o projeto de tal modo que o programa resultante apresentasse o maior número possível de recursos por unidade de área de memória ocupada. É evidente que, na maioria das vêzes, uma tentativa de adicionar mais recursos ao programa sem aumentar a área ocupada pelo mesmo pode trazer problemas no desempenho do programa, tais como queda de velocidade, quebra de estrutura, etc. Em alguns destes casos, como será visto, optou-se por soluções de compromisso, e, em outros, por soluções que dessem ao programa maiores recursos de utilização.

No presente caso, tem-se os seguintes fatores a considerar :

a) baixa capacidade de memória (4K palavras de 8 bits)

b) baixa velocidade de saída dos dados (teletype: 10 palavras por segundo)

c) deficiência do conjunto de instruções no que se refere a:

- manipulação de dados com mais de uma palavra (p/ex, dados em precisão dupla).

- comparação de dados

- extração e inserção de campos dentro de uma palavra (inclusive mascaramentos)

- operações aritméticas

Analisando-se os fatores a) e b), levando-se em conta observações realizadas nos programas aqui descritos, chegou-se às seguintes conclusões:

- O tempo de processamento é desprezível em relação ao de entrada e saída. Isto permite concluir que mesmo se o programa for algumas vezes mais lento, o tempo total de execução será praticamente o mesmo.

- Rotinas otimizadas em relação ao tempo de processamento ocupariam, em certos casos, até cerca de 40% a mais de memória em relação a rotinas equivalentes, otimizadas em relação ao número de palavras ocupadas pela mesma na memória. Levando-se em conta o fator a), percebe-se que, quando a dimensão do programa for comparável à dimensão da memória disponível, pode-se chegar até a uma condição de inviabilidade de implementação do mesmo se se insistir em utilizar rotinas otimizadas em velocidade. Deve-se observar que tais rotinas sejam, em média, poucas vezes mais rápidas, o que já foi dito ser praticamente irrelevante no caso.

- Muitas vezes, na tentativa de se reduzir o espaço

ocupado na memória para executar uma certa tarefa, pode ser encontrada uma organização eficiente dos dados que permita a utilização de rotinas rápidas com baixo gasto de memória. Para isso, deve ser tirado o máximo proveito de características particulares da arquitetura da máquina.

- Deve-se levar sempre em conta que este trabalho é dirigido para um computador pequeno e restrito. Assim sendo, algumas das argumentações aqui apresentadas podem não ser válidas para máquinas maiores, e algumas considerações, decisivas neste contexto, podem tornar-se totalmente irrelevantes se no lugar do Patinho Feio for colocado um computador de parte maior.

CAPÍTULO 2. O MONTADOR

2. O MONTADOR

Neste capítulo é descrito o programa montador desenvolvido para o Patinho Feio, e cuja finalidade é a de permitir que se programe em uma linguagem simbólica próxima de linguagem de máquina deste minicomputador. Parte-se de uma discussão sobre problemas gerais de programação em linguagem de baixo nível, estudando-se a seguir as características desejáveis para o programa montador, ao lado dos problemas enfrentados na implementação das mesmas num computador do porte do Patinho Feio. A seguir, descreve-se alguns detalhes importantes de projeto e de implementação, finalizando com a descrição de alguns recursos de controle do programa e da lógica do montador implementado.

2.1. O conjunto de instruções e a linguagem do montador

2.1.1. O conjunto de instruções: O Patinho Feio possui um conjunto de 56 instruções, descritas quanto ao funcionamento nas referências (1) e (4). Estas instruções podem ser divididas, para efeito dos algoritmos de montagem que são utilizados, em seis grupos, detalhados no apêndice 1:

- instruções de referência à memória;
- instruções de endereçamento imediato;
- instruções de deslocamentos e giros;
- instruções de entrada e saída;
- instruções curtas com operando;
- instruções curtas sem operando.

Todas as instruções pertencentes a um dado grupo, utilizam a mesma rotina de montagem, como será visto em 2.5.3.

2.1.2. Programação em linguagem de máquina: Dado um computador com seu conjunto de instruções, a única maneira de fazê-lo funcionar, se não se dispuser de nenhum outro recurso auxiliar como, por exemplo, um montador em outro computador ("Cross-Assembler"), será programá-lo em linguagem de máquina, isto é, construir uma sequência lógica de zeros e uns, tais que, carregados convenientemente na memória do computador, e a seguir executados, perfaçam a tarefa prevista.

Quando se procede desta maneira, o trabalho de escrever um programa, carregá-lo na memória e em seguida executá-lo é árdua, pois requer muitos cuidados para que o problema que se está atacando seja resolvido corretamente. Assim, dado um problema para ser resolvido em um computador no qual só se dispõe de linguagem de máquina e de um painel, deve-se primeiramente estabelecer um diagrama de blocos da solução do problema, e, em seguida, traduzí-lo para a linguagem de máquina do computador. Como esta tarefa é muito desagradável e trabalhosa para ser efetuada diretamente, convém que o programa não seja imediatamente codificado em linguagem de máquina a partir do diagrama de blocos, mas que se escreva tal programa em uma linguagem intermediária mais acessível, onde cada instrução, que na linguagem de máquina é representada por uma sequência de zeros e uns (que em nada lembram a função que ela deverá executar no programa), será representada por um símbolo, formado de um ou mais caracteres, que de alguma maneira informe ao programador o papel de tal instrução no programa.

Na linguagem simbólica do Patinho Feio, uma instrução qualquer terá no caso mais geral o seguinte atributos (fig.2.1.2.1):

A	B	C	D
INIC	CARI ARM ARM	2Ø CONTA CONTB	(INICIO DO PROGRAMA)

Fig. 2.1.2.1

Exemplo de instruções em linguagem simbólica do montador

A - um endereço, que poderá ser simbólico (optativo);

B - um mnemônico, que indica qual instrução a ser executada;

C - um operando, que completará a função representada pelo mnemônico;

D - um comentário, para efeito de documentação do programa;

Por exemplo, um pequeno programa,escrito numa destas linguagens, poderia ter o seguinte aspecto:

1. carregue a variável X no acumulador (inicialmente X = 16);
2. some 8∅ ao acumulador
3. guarde o resultado em X;
4. pare;
5. desvie incondicionalmente para 1.

Recodificado na linguagem simbólica do Patinho Feio, o programa acima poderá tomar o seguinte aspecto:

```
UM   CAR   X   CARREGA X NO ACUMULADOR
     SOMI  8∅  SOMA 8∅ AO ACUMULADOR
     ARM   X   ARMAZENA O RESULTADO EM X
     PARE      PARA
     PLA   UM  DESVIA INCONDICIONALMENTE PARA UM
X    DEFC  16  VALOR INICIAL DE X = 16
```

Consultando uma tabela de correspondência entre as instruções simbólicas e os respectivos códigos de máquina, e atribuindo arbitrariamente o endereço $1∅∅_{16}$ à primeira instrução do programa, poder-se-á recodificar o programa, agora em linguagem de máquina (os códigos abaixo estão em notação hexadecimal):

ENDEREÇO	CÓDIGO	RÓTULO	MNEMÔNICO	OPERANDO	COMENTÁRIO
1∅∅	41∅9	UM	CAR	X	AC: = X
1∅2	D85∅		SOMI	8∅	AC: = AC + 8∅
1∅4	21∅9		ARM	X	X: = AC
1∅6	9D		PARE		PARE
1∅7	∅1∅∅		PLA	UM	VOLTE P/UM
1∅9	1∅	X	DEFC	16	X (INICIAL) = 16

Este programa, no seu formato definitivo para utilização do montador, poderia ficar com o seguinte aspecto:

```
        ORG    /1ØØ   ESTABELECE ENDEREÇO DE ORIGEM
UM      CAR    X      AC: = X
        SOMI   8Ø     AC: = AC + 8Ø
        ARM    X       X: = AC
        PARE          PARE
        PLA    UM     VOLTE P/UM
X       DEFC   16     X (INICIAL) = 16
        FIM    UM     ESTABELECE FIM DO PROGRAMA
```

As colunas de endereço e de código não aparecem nesta versão.

Observe-se que o programa assim escrito é auto-explicativo, o que não ocorre com o mesmo programa escrito em linguagem de máquina, como se pode notar observando apenas as duas primeiras colunas de penúltima versão.

Como foi visto no exemplo acima, o programa escrito nesta linguagem intermediária pode ter um formato completamente livre, a critério do programador. Se forem estabelecidas algumas regras de sintaxe e padronizações para esta linguagem intermediária, de tal modo que todos os programadores representem com os mesmos símbolos as mesmas operações, terá sido estabelecida uma linguagem de programação para este minicomputador. A esta linguagem dá-se o nome de linguagem do montador ("assembly language").

Desde que a sintaxe da linguagem do montador obedeça fielmente a regras bem definidas, haverá uma relação unívoca entre um programa escrito na linguagem do montador e o programa obtido a partir dele, escrito em linguagem de máquina. Sendo esta montagem algorítmica, poder-se-á escrever um programa relativamente simples que permita a automação desta tarefa de tradução. A este programa dá-se o nome de montador ("assembler").

2.2. A conveniência de um montador

Como foi dito em 2.1., a programação em linguagem de máquina requer inúmeros cuidados para que o programa obtido seja

executado com sucesso. Uma das tarefas mais laboriosas é a da atribuição correta de endereços numéricos aos endereços simbólicos definidos no programa. Isto decorre principalmente pelo fato de que as instruções do computador não ocupam todas o mesmo número de posições de memória, fato que se verifica em todos os computadores cujas instruções não tenham o comprimento uniforme.

Calculados corretamente todos os endereços e montado o programa, este problema surge cada vez que se desejar corrigir o mesmo, nele inserindo, ou dele retirando algumas instruções (por exemplo, quando da sua depuração ou otimização). Nestes casos será sempre necessário recalcular todos os endereços e corrigir todas as referências à memória que houverem sido alteradas pela correção. Além disto, como o comprimento do programa terá mudado, bem como a posição de memória ocupada pelas instruções deverá o programa ser, mais uma vez, carregado na memória integralmente.

Levando em conta que se dispõe apenas de um painel de chaves para a carga de programas, pode-se facilmente verificar que esta tarefa deve ser substituída por outra mais suave e confiável. Se for viável gerar uma fita de papel que possa ser lida por uma leitora de fitas, e que contenha como informações o código de máquina correspondente ao programa que se deseja carregar e executar, o problema estará resolvido. Para isto, será necessário escrever um programa montador, ou em linguagem de máquina, ou então em outro computador.

Optou-se pela alternativa de escrever o montador em linguagem de máquina.

2.2.1 Rotinas auxiliares para a elaboração do montador

Para que o processo da implementação do montador, em linguagem de máquina, se tornasse menos ineficiente, foram escritos, também em linguagem de máquina, alguns programas auxiliares, com a função de atenuar as dificuldades encontradas na operação do sistema:

- um carregador de fitas binárias absolutas, cuja função é a de carregar na memória um programa apresentado em fita

de papel, em formato binário;

- um descarregador de código de máquina para fita de papel, em formato aceito pelo carregador de fitas, cuja finalidade é a de salvar o conteúdo da memória em fita de papel, para posterior utilização;

- um descarregador de memória para o terminal,cuja finalidade é a de informar o estado da memória num certo instante,fornecendo no terminal um mapa com o conteúdo da memória em formato hexadecimal;

- um carregador de memória a partir do teclado do terminal, cuja finalidade é a de evitar o uso do painel de chaves,fornecendo um modo mais cômodo e confiável de carregar programas ou dados na memória, a partir de dados escritos em formato hexadecimal do teclado do terminal.

Com estes novos recursos à disposição (apêndice 2), passou-se a elaborar o montador.

2.3. Definição das Características Gerais do Montador

O primeiro passo para a elaboração do montador foi o de estabelecer algumas regras gramaticais para sua linguagem, bem como escolher os mnemônicos que representariam doravante as instruções de máquina. Além disso, deciciu-se também neste estágio sobre a existência ou não de determinados recursos na linguagem. O resultado desta fase foi o seguinte:

A) o formato de entrada é livre. Um comando genérico consta de campos, os quais são separados obrigatoriamente por brancos, e é terminado pela sequência de caracteres especiais "return", "linedeed";

B) um caráter especial "rubout" em qualquer posição da linha, anula inteiramente esta linha, até que seja encontrada a sequência "return", "linefeed";

C) um asterisco na 1a. coluna signfica que a linha é de comentário;

D) todos os comandos terão basicamente 4 campos:

- campo de rótulos - que começa na primeira coluna, e que é opcional. Isto quer dizer que, se a primeira coluna estiver em branco, o campo dos rótulos está vazio.

Se existir, deve constar de um ponto ou então de uma sequência de letras, de comprimento qualquer, das quais são consideradas, se existirem, apenas as duas primeiras e a última, para efeito de endereçamento simbólico (esta restrição não é uma limitação séria quanto ao número de rótulos, uma vez que se consegue formar, com as 26 letras do alfabeto, segundo a regra descrita , 18278 rótulos diferentes, embora o número máximo de símbolos permitidos em um programa seja de 256).

- campo dos mnemônicos - que se inicia no primeiro caráter não branco encontrado na linha após o primeiro branco e termina no caráter que precede o próximo branco ou "return" da linha. Este campo é obrigatório. São válidos neste campo os mnemônicos definidos no apêndice 1 e que foram inspirados nos da ref. 1. Também neste campo são utilizados para identificação apenas os dois primeiros e o último caráter do mnemônico. Os demais são ignorados.

É este campo que irá definir o comportamento do montador em relação à linha que o contém, estabelecendo se a linha deve gerar duas, uma ou nenhuma palavra de código, se é apenas controle do montador, etc, ou então, no caso de geração de código, definindo qual o algoritmo de montagem a ser utilizado.

- campo dos operandos - este campo está subordinado ao campo dos mnemônicos. Conforme a classe do mnemônico aí encontrado, o campo dos operandos deve ser composto conforme as regras seguintes:

o instruções de referência à memória: exigem operando, que poderá ser:

1) Simbólico puro. Ex:	CAR YA	(qualquer símbolo definido no programa)
2) Simbólico relativo. Ex.:	CAR YA+3	(idem, seguido de deslocamento absoluto)
3) Relativo puro. Ex.:	CAR *+5	(* seguido ou não de deslocamento absoluto)

4) Absoluto.	Ex.:	CAR	/31Ø	(qualquer constante sem sinal ou *-*)
5) Local puro.	Ex.:	CAR	.-2	(.- ou .+ seguido ou não de um digito hexadecimal não nulo)
6) Local relativo.	Ex.:	CAR	.+ -5	(.- ou .+ seguido ou não de um digito hexadecimal não nulo, seguido de deslocamento absoluto)

• instruções de endereçamento imediato: exigem operando que deverá ser uma constante, com ou sem sinal.

Ex.:	CARI	25
	SOMI	+@P
	NAND	-1Ø2
	XOR	-/Ø1

• deslocamento e giros: exigem operando constante, entre Ø e 4 inclusive.

Ex.:	DD	1
	DE	/Ø1
	GEV	+4

• entrada e saída: exigem operando constante. Depois de convertido para binário, o operando tem o significado seguinte: seus 4 primeiros bits indicam o canal utilizado e os 4 últimos o tipo de ação de entrada e saída a ser executada.

Ex.:	FNC	/D2	
	SAL	/B2	
	ENTR	48	(48_{10} = /3Ø)
	SAI	-48	(-48_{10} = /DØ)

• curtas com operando: exigem operando constante, a saber:

1 - Testes dos flipflops Transbordo (T) e Vai Um (V): o operando deve ser interpretado como booleano (Ø ou ≠ Ø) :

Ex.:	SVM	Ø	≡	SVM Ø
	ST	/3	≡	ST 1
	SV	-8	≡	SV 1
	STM	- @P	≡	STM 1

2 - Painel: operando deve ser uma constante entre Ø e 7:

Ex.:	PNL	Ø
	PNL	/5

• curtas sem operando: estas não exigem operando.Se algo for colocado no campo dos operandos, será ignorado.

Ex.:	TRI	*	≡	TRI
	INC	ABCDE	≡	INC
	PARE		≡	PARE

• pseudo-instruções ("pseudo"): cada uma das pseudos exige operando adequado:

NOME, SEGM, SUBR, EXT - Exigem operando simbólico puro — NOME X; EXT DRIVER

ENT - Exige operando simbólico puro ou relativo — ENT ABC; ENT A+5

ORG - No montador absoluto, (2.4.1), esta pseudo exige operando absoluto se for o 1º ORG do programa. Caso contrário, o operando poderá ser absoluto, simbólico puro, simbólico relativo, relativo puro, local puro ou local relativo.

Exceção. Se for local, não poderá ser .+ n.

No montador relocável: (2.4.2), valem as mesmas considerações, exceto que o operando não pode ser absoluto. São permitidos apenas os tipos simbólicos puro ou relativo, relativo puro, e local puro ou relativo desde que não seja .+n.

DEFE
DEFI } - Exigem operandos simbólicos, absolutos, relativos ou locais.

BLOC
DEFC } - Exigem operando constante decimal, ASCII, ou hexadecimal,
COM com ou sem sinal.

EQU - Análogo ao ORG absoluto.

FIM - No montador absoluto (2.4.1), exige operando qualquer.
No montador relocável, (2.4.2), o operando indica endereço de execução se programa principal, ou deve ser zero se for uma subrotina.

- campo dos comentários - no campo dos comentários (opcional) pode figurar qualquer sequência de caracteres que não inclua os caracteres especiais "rubout" "return" ou "linefeed". O campo dos comentários é terminado com a sequência "return""linefeed", a qual também serve para encerrar a linha do comando.

E - qualquer programa deve ser iniciado por um comando de definição de origem. Há quatro destes comandos: ORG, NOME, SEGM e SUBR. Como será visto em detalhes em 2.5.6, o primeiro é utilizado para definir a origem de programas absolutos, ao passo que os demais servem para associar o endereço relativo "zero" à primeira instrução de um programa principal, segmento ou subrotina, respectivamente;

F - qualquer programa deve ser finalizado por um comando de final de programa. Esta é a pseudo FIM, que, nos programas absolutos e subrotinas relocáveis indica apenas o final físico do programa, e, nos relocáveis, associa além disto no caso de progra

mas principais, o endereço de execução do código gerado;

G - os mnemônicos que representam instruções de máquina serão, quando possível, os mesmos utilizados na documentação dos circuitos correspondentes. Se não for possível o uso dos mesmos mnemônicos, os adotados serão tão próximos quanto possível dos originais;

H - levando em conta o pouco espaço (quantidade de memória) disponível para o programa e para os dados, deve-se escolher uma forma de melhorar o aproveitamento da memória para a construção da tabela de símbolos (2.5). O ideal é maximizar o número permitido de símbolos, e, ao mesmo tempo, minimizar o espaço ocupado por eles na tabela.

Para maximizar o número permitido de símbolos em um espaço fixo de tabela, pode-se por exemplo minimizar o espaço de tabela que cada uma delas ocupa. Assim, por exemplo, seria bom que, na tabela, cada símbolo pudesse ser representado por uma única palavra de 8 bits. Uma possibilidade de realizar isto seria a de utilizar símbolos de dois caracteres, sendo o primeiro alfabético e o segundo numérico. O alfabeto, tendo 26 letras, necessita na sua representação interna, de 5 bits, e as seis combinações restantes poderiam ser preenchidas com caracteres especiais. Os 3 bits que restam na palavra seriam preenchidos pela representação binária de algarismos entre Ø e 7.

O número máximo de símbolos é neste caso 256. Uma vantagem desta solução é a seguinte: se forem reservadas 256 posições na tabela, o símbolo não necessitará estar presente na mesma permanecendo na tabela apenas as informações relativas a ele: o símbolo estaria implícito na posição da tabela em que as informações a ele relativas estariam presentes. Além disto, há uma facilidade adicional pela simplicidade da busca das informações: basta que o registrador de índice seja carregado com o símbolo compactado para que se tenha acesso à informação na tabela com uma única instrução indexada.

Uma desvantagem dêste método é clara: os símbolos não são

mnemônicos, tornando portanto mais difícil o acompanhamento lógico de um programa escrito na linguagem do montador usando símbolos assim definidos.

Sempre que para um problema apareçam duas soluções diferentes, uma de fácil implementação mas que representa para o usuário maior dificuldade de programação, e outra de implementação mais elaborada, mas que poupa esfôrço ao usuário, é recomendável que se opte pela segunda desde que isto não venha a causar um problema de insuficiência de área de memória para o restante do programa em questão. Caso isto ocorra, uma eventual solução de compromisso pode ser tentada.

Com base nestas considerações, a idéia de utilização de símbolos de dois caracteres exposta acima foi abandonada, tentando-se um resultado melhor com símbolos de três caracteres alfabéticos, compactados em 15 bits, o que ocupa duas palavras de oito bits, restando ainda um bit para usos eventuais em futuras modificações do programa. Na versão definitiva do montador, os símbolos permitidos são de comprimento qualquer. Internamente, no entanto, são considerados apenas os dois primeiros caracteres e o último. Naturalmente esta solução não é a mais econômica para o sistema, nem a mais cômoda para o usuário, mas foi a que melhor se adaptou às exigências do usuário e do sistema, entre todas as que foram experimentadas;

I - quanto aos tipos de referência à memória permitidos decidiu-se nesta fase permitir, na primeira versão do montador("bootstrap", isto é, versão inicial, com a ajuda da qual o programa definitivo é construído), apenas referências absolutas e simbólicas puras. As referências locais, bem como as relativas ao Contador de Instruções e a rótulos foram introduzidas mais tarde, no montador definitivo;

J - quanto às constantes, permitiu-se declarar, no "bootstrap", constantes decimais com ou sem sinal, hexadecimal sem sinal e ASCII sem sinal. Na versão atual são permitidos, além destes tipos, constantes hexadecimais e ASCII com sinal. As constantes apresentam-se em seis formas diferentes. Em princípio não há restrição quanto a sua amplitude. Entretanto, como a conversão é fei

ta para formato binário de oito bits, o valor final é sempre o número binário formado pelos oito bits menos significativos do número.

FORMATOS DAS CONSTANTES

Sem Sinal:

a) decimais: As constantes decimais sem sinal constam de uma sequência de dígitos decimais (Ø a 9) precedidos e sucedidos por um delimitador.

Ex.: ØØ1; 152; 9999

b) hexadecimais: As constantes hexadecimais sem sinal constam de uma sequência de dígitos hexadecimais (Ø a 9 e A a F), precedidos por um caráter "/" e sucedidos por um delimitador.

Ex.: /ØØ1; /AFF; /13E

c) ASCII: As constantes ASCII sem sinal constam de um caráter especial "@" sudedido por um caráter ASCII qualquer.

Ex.: @1; @P; @@; @Ø

Com Sinal:

d) decimais: As constantes decimais com sinal constam de um - sinal (+ ou -) sucedido de uma sequência de dígitos decimais:

Ex.: +53; -18; +ØØØ1

e) hexadecimais: As constantes hexadecimais com sinal constam de um sinal seguido de uma constante hexadecimal sem sinal:

Ex.: +/135; -/12 : -/A8E

f) <u>ASCII</u>: As constantes ASCII com sinal constam, analogamente, de um sinal seguido de uma constante ASCII sem sinal.

Ex.: - @-; + @ @; - @ƀ; + @1

K - decidiu-se também quais as pseudo instruções que deveriam existir.
Implementou-se, no "bootstrap", as pseudos ORG, BLOC, DEFC e FIM.
Na versão definitiva acrescentou-se (cap. 2.5) NOME, ENT,EXT, EQU, COM, SEGM, SUBR, além de dar maior versatilidade às outras. Assim, permitiu-se, na versão definitiva, ORG com operando simbólico e/ou relativo, DEFC com operandos com ou sem sinal, BLOC com operandos constantes quaisquer (no "bootstrap" o operando de BLOC tinha que ser decimal e sem sinal).FIM significava, no "bootstrap", apenas o final do programa. Na versão final, o seu operando pode significar o endereço de execução, dependendo do tipo de programa.

L - decidiu-se quanto ao número de passos e quanto à ocasião em que as deteções de erro iriam ocorrer.

Diz-se que um programa tradutor tem n passos quando o programa fonte e/ou programas dele derivados são lidos, analisados e modificados n vezes pelo programa tradutor, antes que o programa objeto (traduzido) esteja pronto. É muito comum a confusão entre número de passos e número de fases de um programa tradutor. O número de passos é decorrente da divisão lógica de um programa em partes estanques, que são executadas geralmente na mesma sequência. Eventualmente um passo pode ser tão complexo ou extenso que exija uma subdivisão em partes menores chamadas fases, módulos ou ainda segmentos.Estas fases interagem normalmente umas com as outras, e sua sequência de execução é geralmente uma função do particular programa que está sendo traduzido. Nota-se portanto que um programa tradutor é dividido em fases puramente por razões físicas e não lógicas. Durante o projeto de um programa tradutor, o número de passos pode ser definido "a priori", o que não ocorre com o número de fases, que depende fortemente de decisões tomadas durante a etapa de implementação.

Havendo opção entre um ou mais passos, resolveu-se fazer uma análise de vantagens e desvantagens de cada uma delas. À primeira vista , a opção de um passo único é mais atraente, pois neste caso o programa fonte deve ser lido apenas uma vez, e à medida que isto acontece, o código objeto vai sendo gerado e a listagem sendo feita. Esta opção realmente é interessante para máquinas onde haja possibilidade de se armazenar na memória o programa objeto gerado para que, montado o programa todo, seja executado o código logo a seguir ("assemble and go"). No caso isto é impossível, uma vez que a memória está praticamente lotada pelo montador,não havendo espaço onde o programa montado possa ser guardado.Assim, é necessário gerar uma fita de papel com o programa objeto, em formato carregável. Outro problema surge além disto, em montadores de um passo: é o das referências para diante: cada vez que se referenciar um símbolo ainda não definido, é necessário guardar a posição e a instrução onde foi feita a referência, bem como o deslocamento em caso de referência relativa, e o símbolo referenciado. Isto pode ser feito com relativa facilidade usando-se a técnica de guardar estas informações em listas ligadas. Estas listas seriam consultadas na ocasião da definição do símbolo, quando seria gerada uma fita contendo as informações convenientes para que o carregador ("loader") pudesse preencher corretamente as posições indicadas. Uma vez feito isto, pode-se eliminar a lista, guardando-se apenas o endereço correspondente ao símbolo, para uso em futuras referências. Mais uma vez, a escassez de memória se faz sentir, pois listas ligadas são ávidas consumidoras de memória: cada elemento desta lista deveria conter no mínimo as seguintes informações, no presente caso:

- posição onde foi feita a referência - 12 bits
- qual a instrução - 4 bits
- informação de deslocamento - 12 bits
- apontador para próximo elemento da lista - 12 bits

40 bits = 5 palavras

No caso de montador "assemble and go" esta lista poderia ser, ao

menos em parte, implementada na própria posição de memória onde o programa está sendo guardado: os apontadores e a informação sôbre a instrução poderiam estar nesta área, enquanto os deslocamentos estariam em tabela a parte, e as posições onde foram feitas as referência estariam implícitos na posição dos apontadores.

O esquema de um único passo apresenta também outras desvantagens: se houver um êrro em algum ponto do programa, todo o tempo gasto em montar e listar o programa até este ponto, bem como a fita objeto gerada, seriam perdidos. Poder-se-ia contornar o problema montando-se o programa uma vez sem as opções de listagem e geração de fita objeto, com a finalidade de detetar êrros. Constatada a inexistência de êrros, far-se-ia nova montagem, desta vez permitindo-se as opções de listagens e geração. Como êste procedimento seria rotineiro, haveria necessidade de duas leituras do programa fonte, sempre que se quisesse montá-lo sem correr o risco de perder o tempo de geração de uma listagem e da fita perfurada com o código objeto.

Levando-se em conta êstes fatos, bem como a maior facilidade de programação do montador em dois passos, optou-se por êste último esquema. Assim, no primeiro passo devem ser detetados todos os êrros de sintaxe, gerada a tabela de símbolos, e decidido se a memória pode ou não conter todo o programa. Todas as mensagens de êrro são geradas no primeiro passo, de modo que, se o programa conseguir passar por ele sem nenhum êrro e sem nenhum símbolo indefinido, poderá ser lido pelo segundo passo sem o problema de perda de tempo devido a êrros de codificação. No segundo do passo foram, entretanto, conservadas as deteções de êrro e impressão de mensagens para previnir quanto a possíveis êrros de leitura da fita fonte ou manuseio inadequado dos periféricos pelo operador.

Mensagens de êrro obtidas no segundo passo são, entretanto, apenas avisos ao operador de que o equipamento ou a operação não desempenham bem seu papel. No caso de ocorrência deste tipo de mensagem, basta ler novamente a fita de modo correto para que o programa seja montado adequadamente.

2.4. Definição das Características Externas do Montador

Definidas as principais características do programa montador, i.é., a sintaxe da linguagem de entrada, o número de passos, e os recursos de que se deseja dotá-lo, passou-se ao seu detalhamento.

2.4.1. Características do Montador Absoluto

No montador absoluto, não são permitidas referências a símbolos globais, isto é, externos ao programa. Assim, um programa absoluto deve ser auto-suficiente, não podendo, portanto, depender de rotinas de biblioteca referenciáveis por nome. Isto não é uma limitação séria se se dispuser de tais rotinas já montadas em posições conhecidas de memória, caso em que as referências a elas poderão ser do tipo absoluto (por endereço). Outro modo de contornar a situação é obter as rotinas desejadas em formato fonte, e anexá-las, no instante da montagem, ao restante do programa, caso em que se deve tomar o cuidado de evitar a duplicação de símbolos.

Num programa absoluto não devem comparecer também as pseudo instruções características de programas relocáveis, a saber: NOME, ENT, EXT, COM, SEGM, SUBR.

É no entanto obrigatória a presença de uma pseudo ORG com operando absoluto como primeiro comando do programa, definindo a posição de memória correspondente à origem do código. No decorrer do programa pode-se mudar a origem do código. Neste caso, o operando poderá ser absoluto ou então relativo a um símbolo anteriormente definido.

2.4.2. Características do Montador Relocável

No montador relocável, não são permitidas as pseudos ORG com operando absoluto, pois a alocação da memória deve ser totalmente feita pelo ligador-relocador (ref. 2): um ORG com operando absoluto provocaria a geração do código subsequente em posições absolutas de memória, o que iria tirar do ligador-relocador o controle da alocação de memória. É entretanto obrigatória

a presença, como primeiro comando do programa, de uma pseudo NOME, SEGM, ou SUBR, cuja função é dar à origem do programa um endereço relocável zero e um nome para efeito de geração dos mapas de memória na ocasião da ligação (fase em que são atribuídos endereços absolutos às instruções dos diversos trechos relocáveis de um programa, e construídos, a partir deles, módulos executáveis de código objeto absoluto).

Num programa relocável é permitida a presença de pseudos ENT, EXT, COM, cuja função será de definir pontos de entrada como rótulos globais (isto é, referenciáveis, em outros programas, pelo nome), nomes de rótulos globais definidos em outro programa, e áreas de memória comuns a diversos programas ou subrotinas. Tais pseudos irão gerar informações para o ligador-relocador, permitindo que êste execute a relocação conveniente dos programas referenciados, e aloque convenientemente a memória para os diversos subprogramas e dados.

Como foi dito, a pseudo FIM num programa relocável indica, se se tratar de programa principal ou segmento, que o endereço da execução é o indicado pelo seu operando.

2.4.3 A Sintaxe da Linguagem de Entrada

Com base nas decisões e definições anteriormente estudadas, chegou-se às características sintáticas da linguagem simbólica de entrada cujo programa tradutor é o objeto deste capítulo. Descrição pormenorizada das funções das diversas instruções e pseudo-instruções implementadas, bem como das regras de sintaxe da linguagem encontram-se na ref. 7.

2.4.4 Características do código objeto gerado pelo montador absoluto

A geração do código objeto é executada no segundo passo do montador absoluto, e obedece a algumas regras, que são detalhadas a seguir.

Não havendo espaço na memória para a geração do código "in loco" para um procedimento do tipo "assemble and go", tal código deve ser gerado em algum dispositivo de saída, em um formato carregável. Fixado êste formato pelas especificações do carregador absoluto (ref. 2) chegou-se ao formato de fita objeto que consta de blocos de dados e as informações adicionais seguintes: nº de "bytes" de que o bloco se compõe, endereço do 1º "byte" de dados na memória, os dados propriamente ditos à imagem da memória e um "byte" de teste de soma longitudional.

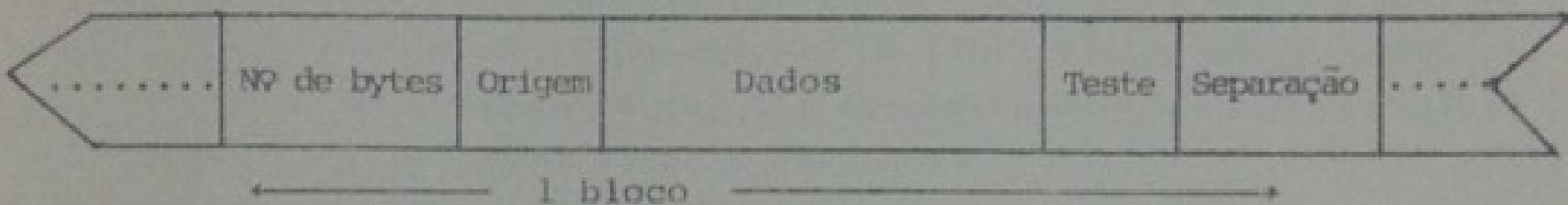



Fig. 2.4.4.1 - Formato lógico de cada bloco de dados na fita objeto absoluta.

Os blocos de que consta a fita absoluta são construídos pelo 2º passo do montador em uma área de memória, a qual deverá ser descarregada para a fita de papel cada vez que uma das seguintes condições ocorrer:

a) preenchimento completo da área de memória: a área deve ser esvaziada para permitir que novos dados aí sejam armazenados;

b) o número de linhas de programa fonte atingiu um múltiplo de 64. Neste caso, se houver um único dispositivo de saída para código objeto e listagem, o número múltiplo de 64 indica final de página, impondo que nesta ocasião seja descarregado o programa objeto, para que a listagem não seja prejudicada;

c) ocorrência de alguma pseudo instrução de mudança de origem (ORG, BLOC). Como o bloco de dados é sempre

preenchido com dados que ocuparão posições contíguas, a ocorrência de uma pseudo instrução de mudança de origem acarreta a geração de dados que geralmente não irão ocupar posições contíguas às anteriores. Pode-se resolver êste problema forçando-se a finalização e descarregamento do bloco que estava sendo preenchido e iniciando-se a construção de um novo bloco;

d) ocorrência de uma pseudo instrução de FIM. Esta pseudo força o descarregamento do último bloco, formado pelos dados que ainda não haviam sido descarregados devido à não ocorrência de nenhum dos eventos citados em a) b) ou c). Como FIM indica final físico do programa , ela deverá forçar tal descarregamento, caso contrário o programa ficaria truncado.

Resumindo: o código construído pelo montador absoluto é gerado à imagem da memória e está pronto para a execução, bastando que seja carregado nas posições convenientes de memória para que possa ser corretamente interpretado e executado pelo computador. Assim, montado um programa numa certa área de memória, êste programa, por referenciar endereços absolutos, poderá não funcionar se fôr carregado em outra região de memória. Por esta razão é necessário que se informe ao programa carregador o endereço em que o bloco de dados deve ser carregado. Além disto, é necessário especificar para o programa carregador o comprimento do bloco, pois como foi visto, os blocos têm comprimento variável. Tem-se, portanto, na fita objeto, 4 informações básicas: o número de palavras de que se compõe o bloco, o endereço em que o bloco deve ser carregado e o bloco de dados propriamente dito. Devido à baixa confiabilidade das operações de entrada e saída em qualquer periférico, é conveniente acrescentar às três informações anteriores mais uma, cuja finalidade é a de, se não eliminar, ao menos reduzir as consequências devidas à incidência de erros de leitura por ocasião da carga do programa. Assim, introduziu-se uma palavra de teste longitudinal de erros de leitura, a qual

é construída tomando-se o complemento de dois da somatória dos números binários correspondentes às instruções que compõem o bloco de dados, às duas palavras de endereço e à palavra que indica o número de dados do bloco. Se, durante a carga do programa, a somatória de todas as palavras lidas, desde o contador de dados até a palavra de teste, não fôr zero, terá sido detetada a ocorrência de um ou mais erros de leitura, o que deverá ser comunicado pelo programa carregador para que o operador possa tentar nova leitura ou investigar a causa do problema. Finalmente, entre um bloco e outro existe uma sequência de quatro zeros, que servem como separadores de blocos, e que são ignorados pelo carregador. Sua finalidade é exclusivamente a de facilitar ao operador a operação de tentar novamente a leitura de um bloco por ocasião de ocorrência de erro de leitura.

2.4.5. Características do Código Objeto Gerado pelo Montador Relocável

No código objeto absoluto, sendo êle gerado à imagem da memória, está implícito que os dados contidos no bloco de código são todos do mesmo tipo, isto é, números binários que deverão ser guardados na memória tais como êles são, não exigindo portanto nenhuma manipulação durante a operação de carga. No caso do código relocável, alguns dados, tais como as constantes e as instruções que não referenciam a memória, terão esta característica. Com as demais instruções nem sempre acontece o mesmo. Em alguns casos a referência é a uma posição fixa de memória, logo o código correspondente será absoluto, e portanto deverá ser gerado como se fôsse uma constante. Mas, na maioria dos casos, uma instrução dêste tipo contida em um programa relocável pode referenciar posições externas (chamadas de rotinas do sistema, por exemplo), endereços internos simbólicos relativos à origem do programa (por exemplo, desvios dentro do programa) ou então endereços internos simbólicos relativos a uma área de dados comum a diversos programas montados separadamente.

Como se pode notar, há 4 tipos diferentes de dados em um programa relocável:

- dados não relocáveis;
- dados relocáveis em relação à origem do programa;
- dados relocáveis em relação à origem de uma área comum de dados;
- dados que indicam referências a posições externas (variáveis globais).

Tem-se portanto que fornecer ao programa ligador-relocador tais informações adicionais quanto à natureza dos dados que êste deverá manipular. O programa não pode ser carregado diretamente por não estar em formato imagem da memória, devendo portanto, ser manipulado e relocado adequadamente (ref. 2) antes que possa ser executado. Apesar da desvantagem de um procedimento adicional intermediário entre montagem e execução, ganha-se muito em flexibilidade, pois isto permite que a manipulação de programas em formato relocável seja vantajosa na grande maioria dos casos em relação ao procedimento de montar novamente o programa fonte utilizando o montador absoluto.

O conteúdo de uma fita objeto relocável (apêndice 3) será portanto condicionado à maneira pela qual êle será tratado pelo programa carregador. Deve-se ter, de acôrdo com as especificações do ligador-relocador (ref. 2) as seguintes informações numa fita objeto relocável:

- um bloco de nome do programa (bloco NOME) onde figura o nome, o comprimento e o tipo (programa ou subrotina);

- blocos que indicam o nome de todas as posições externas referenciadas no programa (blocos EXT), que informam ao ligador-relocador quais rotinas de biblioteca devem ser carregadas junto com o programa em questão;

- blocos que indicam os nomes pelos quais posições internas ao programa serão chamados externamente (bloco ENT) e seus endereços internos;

- blocos de dados, cuja estrutura é análoga à do bloco de dados do montador absoluto, sendo que nêste caso cada dado é acompanhado por uma informação de relocação (informação que associa a cada dado, um dos quatro tipos descritos anteriormente);

- bloco de FIM, onde é guardado o endereço relocável de execução do programa (se fôr o caso de um programa principal ou segmento).

Todos os blocos têm, como primeira informação, o número de palavras, apresentando em seguida, o tipo de bloco, e como última informação, a palavra de teste de erros de leitura, sendo separados entre si por uma sequência de quatro zeros, sem valor lógico.

2.5. Definição das Características Internas do Montador

Estabelecidos os aspectos gerais de funcionamento e os detalhes de algumas alternativas possíveis de solução de problemas relativos ao tratamento do programa fonte, passou-se a considerar os aspectos internos da manipulação das informações nele contidas, visando à obtenção do código objeto a partir de tais informações.

2.5.1. A Representação Interna dos Rótulos

Como foi visto em 2.3., um rótulo, apesar de poder possuir mais de três caracteres no programa fonte, será sempre reduzido ao formato interno padrão de três caracteres, sendo válidos para tal fim os dois primeiros e o último caracteres da sequência de entrada. Caso tenha menos de três caracteres, o símbolo será completado com caracteres especiais "@" à direita até que tenha preenchido o número de três caracteres (fig. 2.5.1.1.).

FORMATO EXTERNO	FORMATO INTERNO
A	A@@
AB	AB@
ABC	ABC
ABCD	ABD

Fig. 2.5.1.1

Exemplo da correspondência entre os formatos externo e interno dos rótulos

O algoritmo de transformação dos rótulos para o formato binário interno é o seguinte:

Sabendo-se que as letras do alfabeto são representadas, no código ASCII sem paridade, por uma sequência de números começando em /41 e terminando em /5A, e que o símbolo se compõe apenas de caracteres alfabéticos, bastarão 5 bits para representá-lo. Convencionando-se representar a letra "A" pelo número $ØØØØ1_2$, "B" por $ØØØ1Ø_2$ e assim subtrair /4Ø do código ASCII correspondente à letra para obter o correspondente compactado nos 5 bits menos significativos da palavra. O caráter "@" será representado por $ØØØØØ_2$, e será injetado artificialmente no caso da ocorrência de um número de caracteres menor que 3 no símbolo original.

Chamando de N_i o número compactado definido acima correspondente ao caráter de índice i (i = 1 correspondente ao primeiro caráter do símbolo, etc), tem-se a seguinte definição do símbolo compactado:

$$C = \sum_{i=1}^{3} N_i \times 2^{5*(i-1)} \qquad \text{em 16 bits}$$

Desmembrando-se em duas palavras de 8 bits tem-se

$$C_1 = \text{int} \quad (C/2^8) \qquad \text{(8 primeiros bits de C)}$$

$$C_2 = C - 2^8 \times C_1 \qquad \text{(8 últimos bits de C)}$$

que definem a representação interna dos rótulos.

EXEMPLO: Para o Rótulo "AB"

A ØØØØ1

B ØØØ1Ø

@ ØØØØØ

Logo C = | Ø | ØØØØØ | ØØØ1Ø | ØØØØ1 |

ou seja: C_1 = | ØØØØ | ØØØØ |

C_2 = | Ø1ØØ | ØØØ1 |

2.5.2. A Organização da Tabela de Símbolos

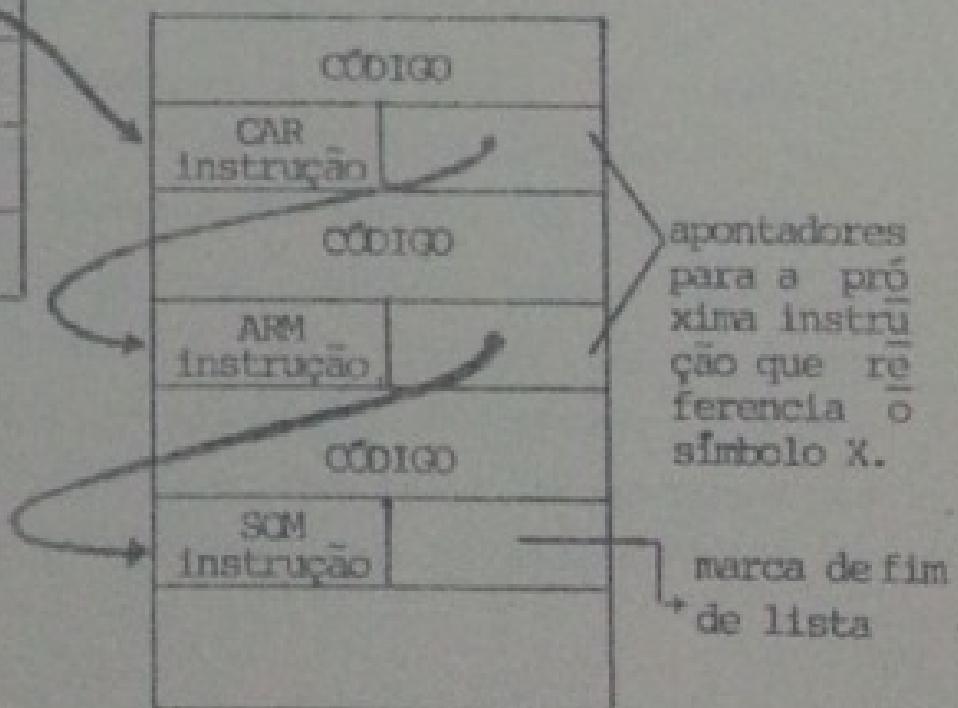
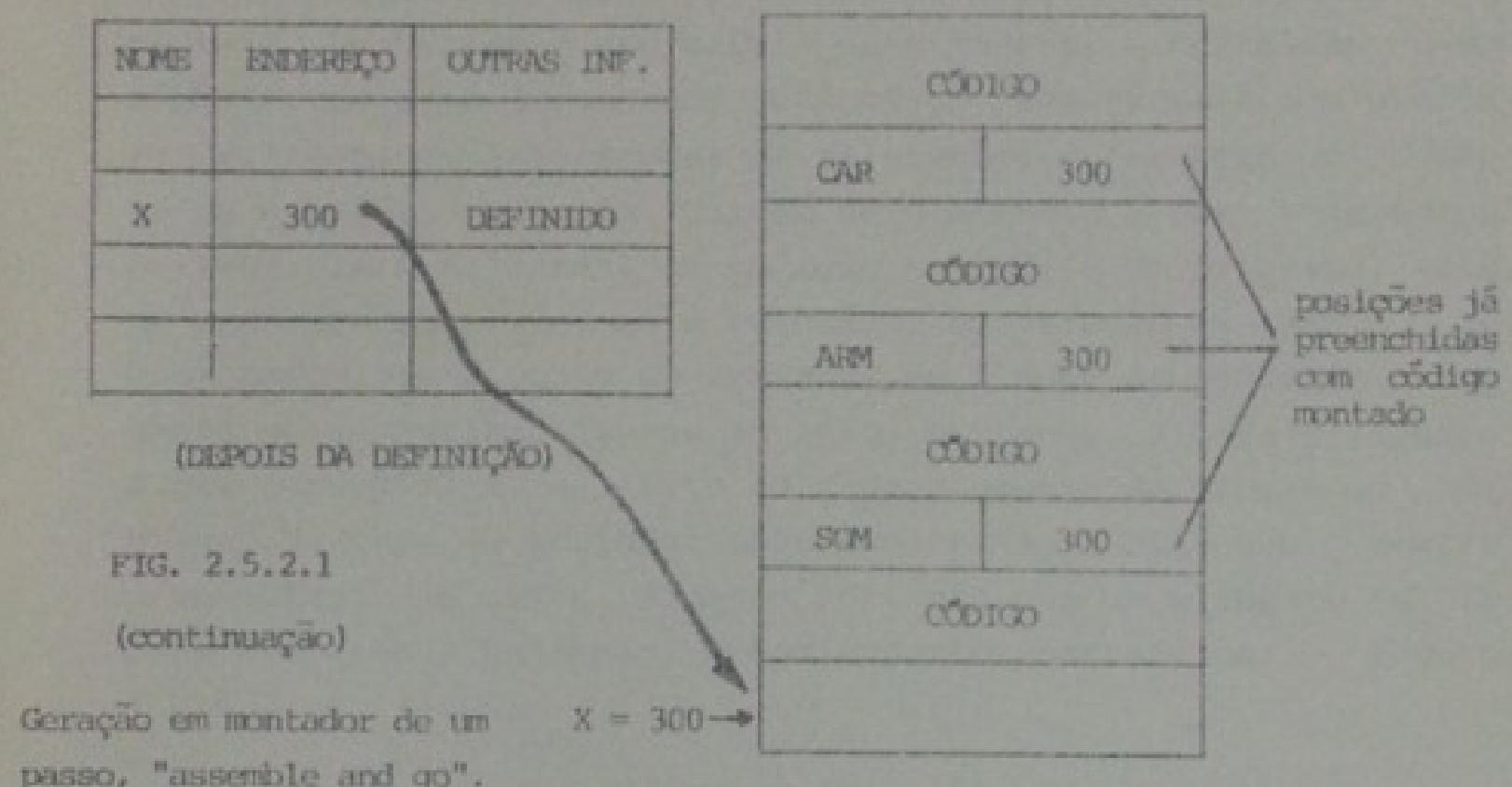
Em um montador, seja ele de um ou dois passos é necessário gerar-se uma tabela de correspondência entre os nomes dos rótulos e os respectivos endereços. Em montadores de um passo, para computadores pequenos, esta tabela será, ao menos em parte, residente na memória no instante da execução do programa que está sendo montado, pois em tais casos, recorre-se normalmente ao uso de posições de ligação ("links"), posições de memória que servem como apontadores para o endereço de memória correspondente ao símbolo que elas representam. Assim, em um montador de um passo para computadores com endereçamento indireto (ref. 8), utiliza-se a técnica de reservar, numa tabela, uma posição de ligação, toda vez que ocorrer uma referência a um símbolo ainda não definido, sendo então substituída a instrução que a ele se refere por uma outra que referencia indiretamente este apontador gerado. Como o endereço destes apontadores é conhecido e como estes vêm acompanhados, na tabela de símbolos por uma marca ("tag") que os identificam como apontadores, a geração

do código poderá ser feita, com relativa facilidade, por êste processo. Assim que o símbolo for definido, a marca de apontador é retirada, substituindo-se o endereço do mesmo pelo endereço real do símbolo. Para computadores com maior capacidade de memória, pode-se escolher outras estratégias de tratamento de símbolos ainda indefinidos, uma das quais seria a de, detetada uma referência a um símbolo ainda não definido, guardar numa lista ligada os endereços onde ocorreram as referências ao mesmo, juntamente com a informação do qual a instrução que o referencia em cada caso. Definido o símbolo, gera-se código objeto nos endereços indicados na lista (técnica de "backtracking"), com as instruções também indicadas, utilizando o endereço real do símbolo para a montagem do código, sem utilizar o recurso do endereçamento indireto. Este processo é indicado quando não se dispõe de endereçamento indireto ou quando se deseja minimizar a área ocupada por informações que não pertencem explicitamente ao programa.

Tira-se da lista, em seguida, todo o conjunto de informações relativo ao símbolo, guardando-se apenas o seu endereço na tabela de símbolos convencional. Em montadores que geram código na própria memória, pronto para a execução ("assemble and go"), a geração da lista ligada pode ser feita "in loco", isto é, a posição da memória onde ocorreu uma referência a símbolo não definido é preenchida com informações sôbre a instrução que aí deve ser montada, e com um apontador para a próxima referência ao mesmo símbolo. O apontador para o primeiro elemento de tais listas fica na tabela de símbolos, e o último elemento deverá conter uma marca identificadora de "fim de lista". (Fig. 2.5.2.1.)

FIG. 2.5.2.1

(continua)

FIG. 2.5.2.1
(continuação)
Geração em montador de um passo, "assemble and go".

Em um montador de dois passos este problema não existe , pois a tabela de símbolos é produto da leitura completa do programa fonte, o qual é relido no segundo passo, e portanto a geração do código objeto utilizará informações provenientes do programa fonte e da tabela de símbolos gerada no primeiro passo.Após a montagem do programa objeto, a tabela de símbolos poderá ser destruída,pois não haverá necessidade de sua presença na memória na ocasião da execução do programa montado. Na fita objeto relocável, entretanto,deverão ser guardadas algumas informações provenientes da tabela de símbolos para posterior utilização,por ocasião da ligação e relocação do programa.Tais informações são guardadas em tabelas a parte: nomes de referências externas, nome do programa, áreas comuns a diversos programas, comprimento do programa,informações sobre o tipo do programa e outras (ref. 2).

Com base nas decisões tomadas em 2.3 e 2.4, e levando em conta a escolha do formato interno dos símbolos, pode-se passar em seguida à elaboração do formato da tabela de símbolos.

As informações que um elemento da tabela deverá conter devem ser as necessárias e suficientes para uma decisão simples e rápida sobre a natureza e sobre os atributos do mesmo. Como foi visto em 2.5.1., o nome do símbolo, uma das informações que o elemento da tabela deve conter, ocupará duas palavras. Como o

endereço a ele correspondente deverá ter 12 bits, haverá necessidade de mais duas palavras, sendo que uma delas será preenchida parcialmente com a parte do endereço correspondente ao símbolo. Um bit será necessário para indicar a definição do símbolo. No caso do montador relocável, são necessários outros dados, chamados informações de relocação, referentes ao símbolo em questão, os quais indicam se o símbolo é uma referência externa, se é relocável em relação ao início do programa, se é um endereço absoluto, se é relocável em relação ao início da área comum, se é ponto de entrada, se é nome de programa, segmento ou subrotina. Tem-se ao todo, portanto, 8 possibilidades o que consumirá mais 3 bits em cada elemento da tabela de símbolos. No caso do montador absoluto, estes 3 bits são sempre preenchidos com zeros. Fica-se portanto com um elemento da tabela ocupando 4 palavras, e com aspecto mostrado na figura 2.5.2.2.

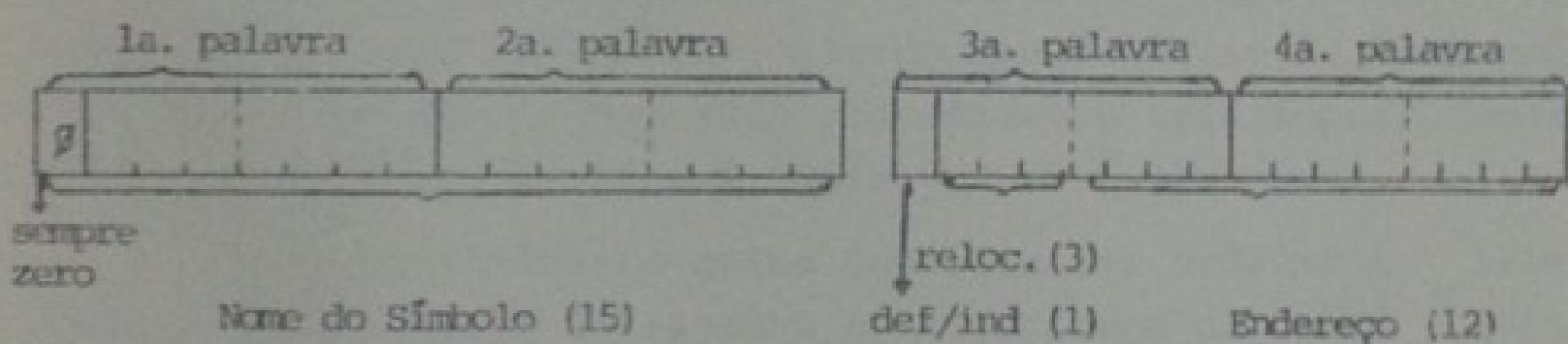



Fig. 2.5.2.2 - Estrutura de um elemento da tabela de símbolos

A ocorrência de um símbolo mais de uma vez no campo de rótulos (múltipla definição) não é considerada na tabela de símbolos, produzindo apenas uma mensagem de erro. O símbolo permanecerá com a primeira definição ocorrida no programa.

Dispondo-se de mais memória e de armazenamento auxiliar em disco ou fita magnética, poder-se-ia incorporar ao elemento da tabela de símbolos mais informações, como número da linha em que ocorreu a definição, e apontador para lista cujos elementos conteriam os números das linhas em que o símbolo foi referenciado no programa. Isto seria opcional, e execu

cem (fig. 2.5.2.3)

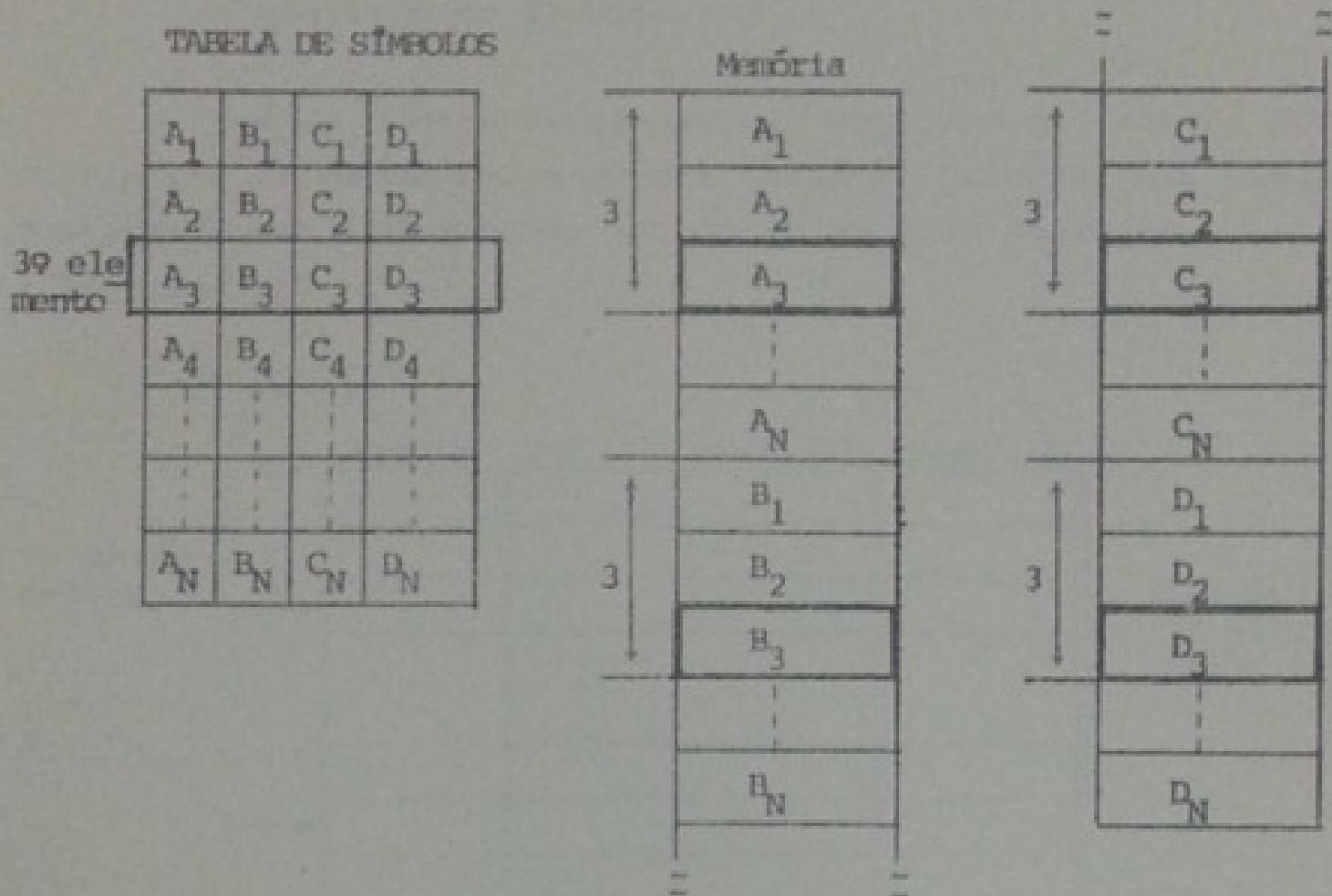



Fig. 2.5.2.3 - Organização na memória da tabela de Símbolos do Montador. Está realçado o terceiro elemento da tabela.

Assim, utilizando-se o mesmo indexador, pode-se ter acesso às quatro palavras, bastando para isto que sejam utilizadas das instruções indexadas (ref. 8) referenciando os inícios das quatro subtabelas resultantes da divisão efetuada.

Com isto são economizados os cálculos de Índices e possíveis cálculos de endereços efetivos caso haja necessidade de dar ao indexador um valor maior que 256.

2.5.3. A Manipulação da Tabela de Símbolos

Passa-se a seguir, com base no que foi concluído em 2.5.2, à definição das rotinas básicas para a criação, manipulação e pesquisa da tabela de símbolos. Como se sabe, a tabela em si contém as seguintes informações lógicas:

- nome do símbolo
- indicador de relocação (tipo de símbolo)
- indicador de definição
- endereço do símbolo

Como o número de elementos contidos na tabela não é constante, há necessidade de uma variável externa que indica quantos são os elementos da tabela em um dado instante. (fig. 2.5.3.1)

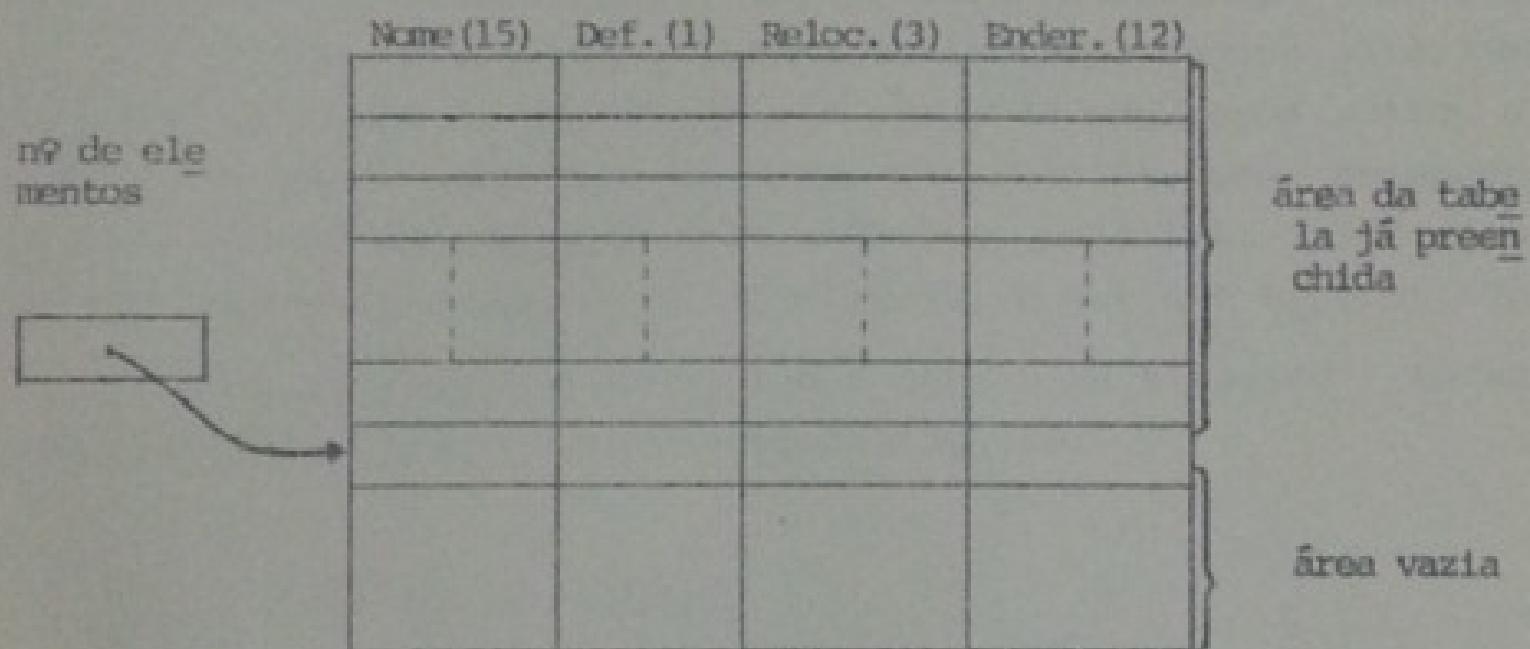



Fig. 2.5.3.1 - Esquema Lógico a Tabela de Símbolos

Três problemas a resolver se apresentam ao se escolher êste esquema para a tabela de símbolos:

a) Tem-se um símbolo, obtido a partir do programa fonte, e deseja-se saber se êle está na tabela, bem como a sua posição em caso afirmativo;

b) Sabe-se que o símbolo não está na tabela, e deseja-se acrescentá-lo aos já existentes;

c) Sabe-se que o símbolo está na tabela, e deseja-se conhecer ou modificar seus atributos.

O problema a) ocorre sempre que um símbolo fôr deteta do no campo dos operandos de instruções de referência à memó ria, ou então no campo dosrótulos.

O problema b) ocorre quando, no caso anterior, o sím bolo não for encontrado na tabela. Neste caso é necessário que se inclua o novo símbolo na mesma. O campo de definição deverá ser feito "definido" caso o símbolo compareça no campo de rótu los, e "indefinido" no caso contrário.

O problema c) ocorre sempre que um símbolo apareça no campo dos rótulos, tendo aparecido anteriormente no progra ma como operando.Neste caso, tal símbolo já está na tabela, mas indefinido, desejando-se torná-lo definido, sendo para isso modificadoso campo de definição e o campo de endereços. Caso o símbolo já tenha aparecido no campo dos rótulos, uma mensagem de erro deverá ser enviada pelo montador, pois isto significa uma tentativa de redefinição do símbolo, o que não é permitido.

As rotinas que manipulam a tabela de símbolos foram implementadas conforme a descrição a seguir.

1. Rotina de Pesquisa

A rotina TAB, de pesquisa na tabela de símbolos,tem como entrada o símbolo a pesquisar, compactado em duas palavras de oito bits. Sua função é a de varrer a tabela, comparando ca da símbolo nela encontrado com o símbolo fornecido, e retornan do com duas informações:

a) se encontrou ou não o símbolo na tabela;

b) apontador para a posição da tabela onde o símbo lo foi encontrado, ou para a primeira posição livre da tabela, se o símbolo não foi encontra do.

2. Rotina de Inclusão

A rotina COLOC, de inclusão de um novo elemento na tabela de símbolos, limita-se a utilizar os resultados da roti

na de pesquisa, que deve ser chamada previamente colocando o novo elemento na primeira posição livre de tabela de símbolos, e ligando o bit de indefinição do símbolo. Esta rotina fornece também uma mensagem de erro no caso de a tabela estar totalmente preenchida.

Em uma versão do montador com mais memória disponível, poder-se-ia incluir nesta rotina uma ordenação alfabética dos símbolos, para efeito de maior facilidade de utilização da listagem da tabela pelo usuário.

3. Rotina de Tratamento geral dos Símbolos

A rotina LABEL, de tratamento dos símbolos, tem por finalidade construir a tabela de símbolos no primeiro passo do montador. Para isso, analisa cada linha do programa fonte, extraindo do mesmo todos os símbolos, classificando-os , calculando quando necessário seus endereços e montando com estas informações a tabela de símbolos. O procedimento é o seguinte:

a) Varre-se a linha, extraindo-se e compactando-se o rótulo, se êste existir. A rotina de compactação se encarrega de detetar invalidade de rótulos;

b) Chama-se a rotina TAB (de pesquisa) se o rótulo existir;

c) Se o rótulo for encontrado na tabela,verifica-se se já estava definido, caso em que uma mensagem de erro é fornecida.Se não estava definido, atribui-se ao campo de endereços o valor do contador de instruções corrente e define-se o símbolo;

d) Se o rótulo não foi encontrado, chama-se a rotina COLOC, definindo-se a seguir o símbolo;

e) Tratado o campo dos rótulos, verifica-se se a instrução apresenta operando simbólico.Em caso

afirmativo, pesquisa-se êste símbolo na tabela, e se não for encontrado, chama-se a rotina COLOC, que o acrescentará aos demais, com o bit de indefinição ligado.

4. Rotina de Listagem e Teste de Consistência da Tabela de Símbolos

A rotina KONSIST, que testa a consistência de tabela de símbolos, deve ser chamada quando fôr detetado o final físico do programa, isto é, o aparecimento do pseudo FIM. Sua função é a de analisar um a um os elementos de tabela de símbolos, fornecendo mensagem de erro sempre que algum símbolo da tabela aparecer com o bit de indefinição ligado. Além disso, a rotina testa se o programador deseja uma listagem da tabela de símbolos, fornecendo, em caso afirmativo, para cada elemento da tabela, uma saída impressa do seu nome, agora descompactado e convertido para o formato padrão de três caracteres, ao lado do endereço que consta da tabela, e das informações de relocação (estas apenas no caso de programa relocável). No final, é impresso o número de símbolos indefinidos - encontrados na tabela.

Todas estas rotinas são utilizadas no primeiro passo do montador. O segundo passo utiliza apenas a rotina TAB de pesquisa, e não altera o conteúdo da tabela de símbolos , obtendo dela apenas as informações colhidas durante o primeiro passo.

2.5.4. A Organização e a Manipulação da Tabela dos Mnemônicos

Definidos em 2.3 como símbolos de três caracteres, os mnemônicos ocupam, a exemplos dos rótulos, depois de compactados, duas palavras cada um.

Representando operações, comandos ou instruções, os mnemônicos podem ser classificados de acôrdo com a sua função. Assim pode-se agrupar os mnemônicos em conjuntos menores cujos elementos têm todos as mesmas características sintáticas e semânticas. Com isso, constatada a validade do mnemônico e des

coberto o grupo a que ele pertence, pode-se, por inspeção das características de seu grupo, decidir qual será a ação a tomar em relação ao respectivo campo de operandos ou na geração do código objeto.

Os grupos de mnemônicos, em número de sete, são os seguintes:

- referências à memória (grupo 1)
- imediatas (grupo 2)
- curtas sem operando (grupo 3)
- deslocamento e giros (grupo 4)
- entrada e saída (grupo 5)
- curtas com operando (grupo 6)
- pseudos (grupo 7)

Desta maneira, instruções do mesmo grupo terão tratamentos análogos, sendo tal tratamento diferente do que receberá uma instrução de outro grupo. Logo, a informação do grupo ao qual a instrução pertence é muito importante para as rotinas de tratamento de rótulos e para a montagem do código objeto.

Há diversas alternativas de construção das tabelasde mnemônicos, todas dirigidas para uma pesquisa rápida, e contendo maior ou menor número de informações sobre a ação do montador em relação ao mnemônico. Um método eficiente parauma busca rápida é organizar tabelas para busca logarítmica (refs. 7,9). Encontrado o mnemônico, pode-se ler na posição correspondente de uma outra tabela, as informações a ele relativas e que poderia incluir a informação sôbre o número do grupo a que a instrução pertence, além de outros detalhes sôbre a instrução específica, como por exemplo o seu código de operação, se a instrução precisa ou não de operando, se é longa ou curta, etc. Quanto mais informações esta segunda tabela contiver tanto maior será o espaço de memória por ela ocupado,

porém será mais rápido e mais simples obter todas as informações de que se necessita para o tratamento do programa fonte. Outros métodos mais complicados para pesquisa de mnemônicos existem , porém, sua grande maioria exige memória adicional para tabelas - auxiliares, para apontadores e mesmo para o algoritmo de busca, o que não os torna aconselháveis no presente caso.

A tabela de mnemônicos foi organizada para busca linear por motivos de simplicidade do algoritmo de busca. Assim, adotou-se a opção de obter as informações sôbre a instrução na posição em que o mnemônico foi encontrado. Isso leva a uma economia de memória pois dispensa tabelas adicionais e algoritmos complicados.

Em vista disso e levando-se em conta que esta tabela não varia no decorrer do processamento, decidiu-se ordená-la de tal maneira que os grupos mais usados fossem encontrados mais rapidamente na busca linear. Assim, o primeiro grupo é das instruções de referência à memória, e o último, o das pseudos. Com isto diminui-se o tempo de busca de um mnemônico na tabela. Uma segunda tabela, a de apontadores do início das subtabelas de instruções do mesmo grupo, tem como finalidade estabelecer os limites de tais subtabelas, permitindo um cálculo simples do número da tabela a que o mnemônico pesquisado pertence. Obtido êste número, indexa-se com êle uma terceira tabela, a dos atributos do grupo . Esta tabela contém as seguintes informações: PSEUDO ou não, LONGA ou curta, OPERANDO ou não, NUMÉRICO ou não, e NÚMERO da subtabela a que o mnemônico pertence. Estas informações foram organizadas em formato decodificado, isto é, cada bit representa direta e independentemente a variável a ele correspondente para maior facilidade de teste, e agrupadas em uma palavra para cada grupo de mnemônicos. Na fig. 2.5.4.1 está , esquematicamente,a organização de tabela de mnemônicos adotada.

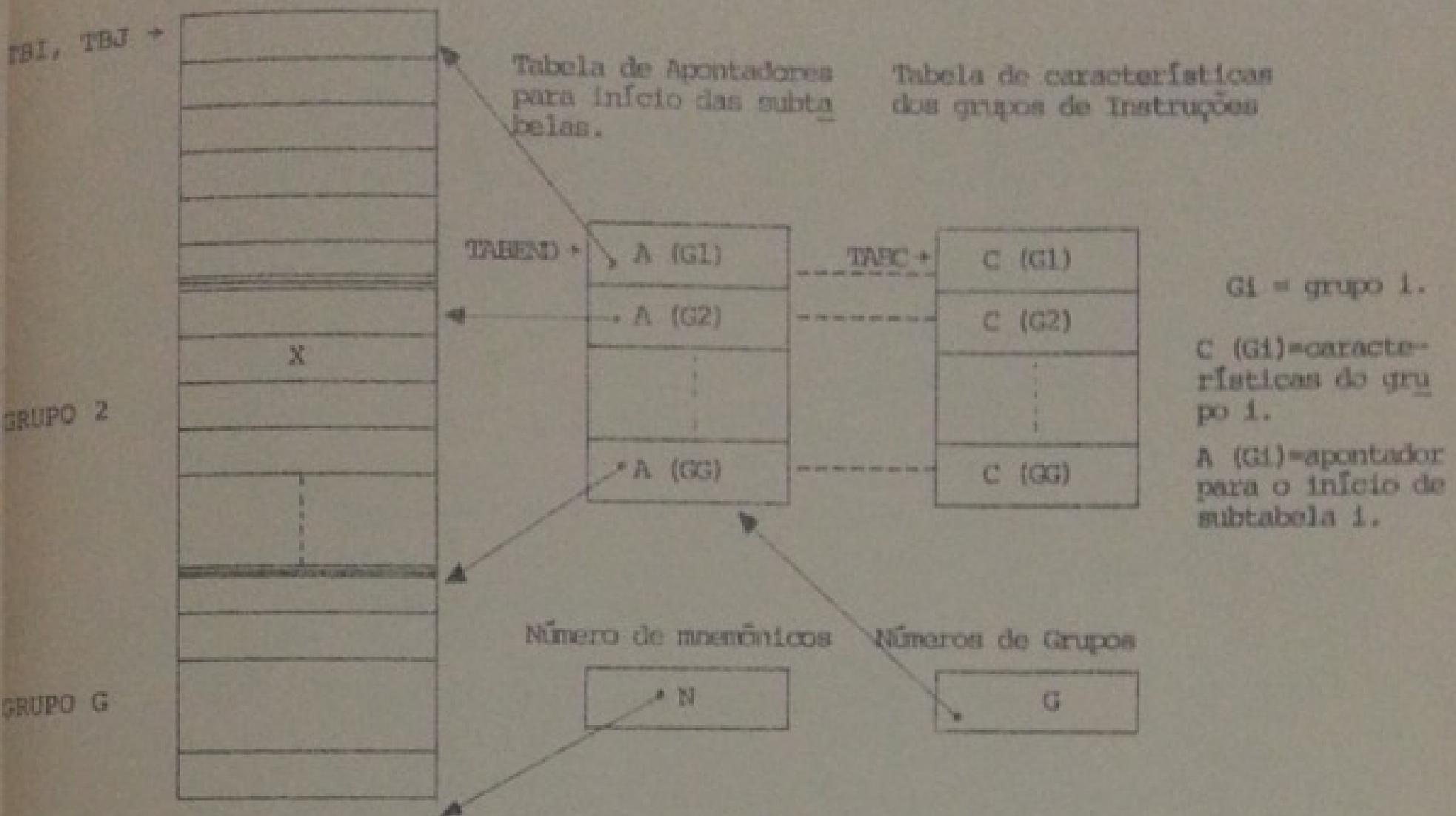



Fig. 2.5.4.1 - Organização da Tabela de mnemônicos

A rotina MNEM é a responsável pela pesquisa e identificação dos mnemônicos, constituindo a parte central do montador. No primeiro passo, sua função é basicamente de pesquisa e de deteção de mnemônicos inválidos e de auxiliar na construção de tabela de símbolos, mas no segundo passo sua participação se faz presente na geração do código objeto. Simplificadamente, o algoritmo utilizado para implementar a rotina MNEM é o seguinte (fig. 2.5.4.1):

a) compara-se o mnemônico a pesquisar com cada um dos elementos da tabela de mnemônicos, até que ele seja encontrado na posição i da tabela ou então que a tabela seja esgotada (neste último caso, uma mensagem de erro é fornecida e o controle é devolvido ao programa que chamou a rotina MNEM);

b) verifica-se, por comparação com os elementos da tabela de apontadores TABEND, a que intervalo, ou seja, a qual subtabela, o mnemônico pertence. Isto é feito pro

curando-se TABEND (j) e TABEND (j+1) tais que TABEND (j) $\leq$ i < TABEND (j+1), caso que o grupo (subtabela) ao qual o mnemônico pertence será j;

c) As informações a respeito do grupo j de instruções, ao qual o mnemônico em questão pertence, são obtidas imediatamente na posição j da tabela TABC de características dos diversos grupos de instruções.

Observe-se que a tabela de mnemônicos é subdividida em duas tabelas, TBI e TBJ porque cada elemento da mesma ocupa 16 bits (a exemplo dos rótulos).

2.5.5. A Representação Interna das Constantes

Como foi visto em 2.3, as constantes podem assumir diversas representações externas:

- com ou sem sinal
- hexadecimal, decimal ou ASCII

Para transformar constantes, escritas em uma das possíveis seis combinações acima, para o formato binário interno em complemento de dois, foi elaborada uma rotina de conversão geral, (CONVERT) que identifica o tipo da constante e em seguida chama rotinas particulares de conversão, uma para cada tipo de dado. As rotinas DECBIN e HEXBIN, que convertem constantes sem sinal, escritas em decimal e hexadecimal, respectivamente, para binário, utilizam-se da definição da representação de um número em uma base dada:

Sendo $A_n, A_{n-1}, \ldots, A_o$ os dígitos que representam um número N (sem sinal) na base B, a representação de N na base B será $A_n A_{n-1} \ldots A_1 A_0$ e o valor de N será dado por

$$N = \sum_{i=o}^{n} B^i A_i$$

Desta fórmula decorre imediatamente o algoritmo de conversão implementado:

a) Posicionar N em zero; Apontar para A_n

b) Se o dígito mais à direita (A_o) já foi manipulado , terminou a conversão (isto é verificado analisando--se o caráter que segue o último dígito tratado, verificando-se se o mesmo é ainda um dígito ou não)

c) Caso contrário, $N \leftarrow B \times N$ + dígito apontado

d) Voltar para b) apontando o próximo dígito

Como não se tem instrução de multiplicação, e como os dados a converter normalmente são endereços, ocupando portanto duas palavras, foi necessário implementar pequenas rotinas específicas para efetuar multiplicação por 1Ø e por 16, e soma em dupla precisão. Foram usados tais algoritmos específicos no lugar de uma rotina geral de multiplicação por serem muito mais econômicas e eficientes para esta aplicação particular (único ponto do montador onde se faz cálculos aritméticos). Multiplicar por 16 equivale, em aritmética binária de complemento de 2, a deslocar o número para a esquerda de 4 bits. Multiplicar por 1Ø equivale a multiplicar por 8 e somar o resultado ao dôbro do número. Como multiplicar por 2 é o mesmo que deslocar para a esquerda de 1 bit, e multiplicar por 8, deslocar de 3 bits, então uma rotina DED de deslocamento à esquerda em dupla precisão, chamada várias vêzes na sequência conveniente, e associada à rotina SOMD de soma em dupla precisão é suficiente para que se possa implementar a subrotina MDD de multiplicação por 1Ø em dupla precisão. O uso das rotinas MDD, SOMD, e DED permitem a construção das rotinas de conversão DECBIN e HEXBIN, que implementam o algoritmo descrito. Executada a conversão, retorna-se para a rotina CONVERT, a qual se encarrega de acertar o sinal do número convertido.

Deve-se observar que a constante decimal ou hexadecimal vem escrita em caracteres ASCII, sendo necessário portanto uma pequena conversão prévia para BCD - (binary-coded decimal) dígito por dígito. Mensagens de erro são fornecidas sempre que for detetado algum dígito ilegal.

No caso de a constante a converter ser ASCII, não haverá necessidade de manipulação das informações da linha, exceto quanto ao sinal. Por isso não foi escrita uma subrotina para o tratamento de tais constantes, sendo êste tratamento executado na própria rotina CONVERT.

As informações em formato interno, apesar de convenientespara o computador, não o são para o usuário. Por isso , ao lado das rotinas de conversão do formato externo para o interno, deve-se ter rotinas que fazem a conversão oposta. No caso, foi implementada uma rotina de conversão binário-hexadecimal CONVHEX, a qual converte um número binário de 8 bits para caracteres hexadecimais, imprimindo-os no dispositivo de saída. Esta rotina é utilizada nas listagens do código objeto, da tabela de símbolos e dos endereços associados às instruções, na listagem do programa fonte.

Os algoritmos de conversão binário-hexa e binário-decimal podem ser resumidos na seguinte descrição :

Seja B a base em questão, e N o número binário a converter. Suponha-se conhecido "a priori" que o número N vai ter no máximo n+1 dígitos na base B. Nestas condições:

a) Posicionar $i \leftarrow n$

b) Fazer $A_i \leftarrow int\ (N/B^i)$ obtendo A_i em BCD

c) Fazer $N \leftarrow N - A_i \times B^i$

d) Fazer $i \leftarrow i - 1$. Se i for negativo, passar para o passo e; caso contrário, voltar para o ítem b.

e) Converter os A_i ($i = 0,1,2...n$) para ASCII e imprimir.

Levando em conta que não se dispõe de instruções de multiplicação nem de divisão, vê-se que não é possível a implantação imediata do algoritmo. Por isso, foi usado o recurso de guardar B^i em tabelas, e de efetuar a divisão por subtrações sucessivas de B^i. Para subtrair em dupla precisão, utilizou-se a rotina SOMD de soma em dupla precisão, associada com uma rotina de complementação de dois em dupla precisão, CMPD. Êste processo, apesar de demorado e ineficiente, permite a implementação do algoritmo em um espaço de memória bastante reduzido, o que não ocorreria no caso de se programar um algoritmo geral de divisão, que seria utilizado apenas nêste ponto do montador.

2.5.6. As Pseudo Instruções

O conjunto das pseudo instruções disponíveis num montador define uma grande parte da versatilidade do mesmo. Pseudo - instruções têm, via de regra, tratamento individual, o que exige uma disponibilidade razoável de memória no caso de se desejar dotar o montador de muitos recursos adicionais. Por essa razão, decidiu-se implementar, na primeira versão do montador, apenas as indispensáveis, acrescentando-se novas pseudo instruções na medida do possível, tendo sempre em vista a quantidade de memória disponível e a vantagem da adição da nova pseudo instrução. Dessa maneira, a primeira versão do montador apresentou apenas as pseudo instruções ORG (para definir a origem do programa), DEFC (para definir uma constante), BLOC (para definir área de dados) e FIM (para definir o final físico do programa fonte). Além disso, tais pseudo instruções tinham formato inflexível para facilidade de implantação rápida. Assim, ORG admitia apenas operandos constantes sem sinal, BLOC admitia apenas operando decimal e FIM servia apenas para finalizar o programa. Implantada a primeira versão do montador, obteve-se com ele um recurso extra para a programação, que fôra até êste ponto feita totalmente em linguagem de máquina. Passou-se a escrever o restante do montador na sua própria linguagem, conseguindo-se com isto reescrever e melhorar uma grande parte do mesmo, dando-lhe característica de montador relocável , e também acrescentando-lhe novos recursos, entre os quais as novas pseudo instruções EQU, NOME, ENT, EXT, DEFE, DEFI, COM, SEGM, SUBR.

Detetada a presença de uma pseudo instrução,desvia-se pa ra a respectiva rotina de tratamento. A seguir,descreve-se o funcionamento e o tratamento de cada pseudo instrução implementada.

2.5.6.1. A Pseudo ORG

Em um programa escrito em linguagem de baixo nível, isto é, linguagem na qual o programador precisa preocupar-se com os detalhes da máquina com a qual está trabalhando, há necessidade de se informar ao montador qual a posição de memória a partir da qual se deseja que o programa seja montado. Evidentemente é disso que depende a informação necessária à montagem das instruções de referência à memória, uma vez que o endereçamento é simbólico e que a cada símbolo é associada um posição de memória.

Portanto, é necessário que se estabeleça em qual posição de memória se deve iniciar a montagem. Isto pode ser implementado facilmente com a escolha "a priori" de uma posição fixa de memória, e criando-se um apontador para esta posição no início da execução do montador. Esta solução é, entretanto, inflexível, exigindo para a modificação de uma posição de memória qualquer, que se monte novamente todo o programa.

Contorna-se esta situação com facilidade criando-se uma pseudo instrução que modifica o apontador para a posição de memória onde se deseja montar o código. Esta pseudo operação, a de ORIGEM do código, poderia ter outras utilidades que não a de apenas definir a posição da primeira palavra do código. Por exemplo, reservar posições de memória para área de trabalho, sobrepor um novo código a outro já montado, etc.Para isto, é desejável que o operando desta pseudo instrução seja o mais flexível possível. Procurou-se por isso generalizá-lo como uma referência qualquer à memória, evidentemente com algumas restrições, que serão vistas adiante.

Deverá ser o primeiro comando no caso de um programa absoluto. Isto ocorre porque é necessário que se estabeleça de início se o programa vai ser absoluto ou relocável e, caso seja absoluto, deve-se indicar a partir de qual posição de memória se deseja montá-lo. Observe-se que nestas circunstâncias é necessário que o operando indique uma posição absoluta de memória, conhecida, não podendo ser, portanto, referência simbólica

pura ou relativa, nem uma referência local, uma vez que, sendo êste o primeiro comando do programa, tais referências estariam indefinidas.

Uma variante para a opção de usar ORG como 1º comando seria estabelecer um valor padrão "default", a ser utilizado no caso da não especificação explícita da origem. Neste caso, haveria a necessidade de criar uma nova pseudo para indicar apenas se o programa é absoluto ou relocável. Novamente, ter-se-ia a opção de adotar um "default" que admitisse ser o programa relocável (ou absoluto com uma origem fixa), no caso da omissão de tal pseudo.

Ainda no caso de um programa absoluto, é interessante que se possa definir, a qualquer momento no programa, que o trecho a seguir deva ser montado a partir de uma nova origem, mediante a utilização da pseudo ORG. Nêste caso, pode-se admitir referências absolutas, relativas, simbólicas e locais, uma vez que é possível nêste caso, que tais referências tenham sido definidas anteriormente. Deve-se notar que, caso a referência não seja absoluta, deverá ela relacionar-se obrigatoriamente com posições já definidas de memória, pois se isto não ocorrer, torna-se, embora possível, bastante complicada a manipulação desta pseudo, da tabela de símbolos e da geração do código, o que torna inviável a elaboração de tal tratamento nas atuais condições. Sendo totalmente dispensável na grande maioria dos casos, êste recurso foi abandonado.

Num programa relocável, é conveniente que se possa definir uma nova origem para o código a qualquer altura do programa. Analogamente ao caso do programa absoluto, tal origem definida pela pseudo instrução ORG deve ser, nêste caso, uma posição conhecida de memória, relativa ao início do programa.

Sendo sempre relativa a origem permitida em programas relocáveis, a pseudo ORG não poderá ser o primeiro comando de tais programas, pois sendo obrigatoriamente relocável (simbólico, relativo ou local), seu operando não teria sentido, pela não ocorrência prévia da definição de tais referências.

2.5.6.2. As pseudos NOME, SUBR, SEGM

Quando se trabalha com programas relocáveis, é comum que um mesmo programa ou subrotina tenha mais de um ponto de acesso global. Assim, se o programa fôr parte de um arquivo de programas relocáveis, como é o caso de uma biblioteca de subrotinas, é conveniente que ele tenha um nome pelo qual possa ser identificado. Tal nome teria,além disso,a função de representar simbolicamente o endereço da primeira palavra do código objeto do programa, endereço êste em relação ao qual as referências relocáveis seriam relativas.

Para definir o nome de um programa relocável, criou-se as pseudos SUBR, NOME e SEGM cujas características são as seguintes:

1. São mutuamente exclusivas, pois, definido o tipo de programa, e sendo êste único, a ocorrência de mais de uma destas pseudos ao mesmo programa será uma tentativa de redefinição do tipo do mesmo;

2. Não deverão estar presentes em programas absolutos , pois sua ocorrência, além de atribuir um nome à origem relocável do programa, serve também para indicar ao montador que o programa em questão é relocável;

3. Deverão ser o primeiro comando de um programa relocável, para forçar que o programa tenha uma identificação.

(Uma alternativa para esta imposição seria a de permitir que um nome "default" pudesse ser atribuído ao programa. Esta possibilidade pode criar problemas, no caso de existir um sistema operacional no qual o programa objeto faça parte na qualidade de arquivo. A tentativa de dois usuários diferentes guardarem dois arquivos relocáveis com o mesmo nome poderia criar confusões para os proprios usuários ou para o operador. Assim, seria conveniente que o sistema operacional pudesse atribuir, ao nome do

arquivo, não apenas o nome fornecido pelo usuário, mas também alguma identificação que ligue o arquivo ao usuário que o gerou).

4. Seu operando, deverá ser somente do tipo simbólico puro, pois não teria sentido como nome de arquivo se não o fosse.

A) - A Pseudo "NOME"

O funcionamento da pseudo NOME é o seguinte:

1. No passo 1, o resultado do tratamento recebido por esta pseudo será o de criar um elemento da tabela de símbolos, correspondente ao nome do programa. Além disso, sua presença caracterizará o programa como relocável.

2. No passo 2, seu tratamento será o de gerar na fita objeto, relocável, um bloco de NOME, no qual estão resumidas as características do programa, tais como comprimento, tamanho da área de variáveis comuns a diversos programas e informação sôbre o fato de o código ser um programa principal. Estas informações não são todas provenientes do operando da pseudo NOME, mas sim coletadas durante a execução de todo o primeiro passo do montador, para utilização nesta ocasião.

   Poder-se-ia juntar algumas destas informações no operando da pseudo NOME. Por exemplo, além do nome do programa, poder-se-ia declarar aí algumas outras informações como por exemplo o tamanho da área de variáveis comuns, o fato de usar ou não rotinas do sistema para o atendimento de interrupção, o tipo do programa (subrotina,programa principal, segmento) etc.

   Por motivos de simplicidade de tratamento, resolveu-se manter a política de coletar tais dados durante o primeiro passo, e criar outras pseudos para identificar subrotinas e segmentos.

3. Na fase de relocação, o bloco de NOME gerado no passo 2 fornece ao ligador-relocador as informações citadas para que possa ser gerado o código absoluto correto para o programa em questão. Além disso, tais informações poderão ser utilizadas pelo sistema operacional para gerar uma lista de atributos que dizem respeito ao programa, no dicionário dos arquivos do sistema.

B) - As Pseudos SUBR e SEGM

Sintática e Semanticamente semelhantes à pseudo NOME, a função destas duas pseudos será a de gerar, no bloco de nome do código objeto, todas as informações descritas para a pseudo NOME, sendo que no caso de SUBR, o programa será identificado como subrotina, e no caso de SEGM, como segmento. O tratamento do programa objeto gerado em cada caso, está descrito na ref. 2.

2.5.6.3. A Pseudo DEFC

Ao se escrever um programa, na maioria das vezes deseja-se criar, em algumas posições de memória, constantes, valores iniciais para alguma variável, ou tabelas com conteúdo conhecido. Para isso é muito interessante a existência, no montador, de um comando que gere, em tais posições de memória, os números binários correspondentes às constantes desejadas. É mais cômodo para o programador que tais constantes possam ser escritas da maneira mais adequada para a ocasião. Pode-se classificar os dados em diversos tipos, de acordo com seu aspecto externo: dados inteiros escritos em binário, octal, hexadecimal, decimal, ASCII, dados em ponto flutuante, em precisão múltipla, e assim por diante.

Sendo o conjunto das rotinas de conversão uma das partes mais trabalhosas e extensas do montador, convém que se escolha de todo este vasto conjunto de possibilidades, o subconjunto que preencha o maior número de requisitos com a menor área

ocupada de memória.

Assim, passou-se à escolha das bases em que se poderá escrever as constantes. Sendo o computador utilizado um computador binário, é conveniente que se use como base de numeração uma potência de 2 tal que a constante seja representável eficientemente, isto é, de maneira facilmente legível e compacta. Para êste fim, as potênciasde 2 que melhor se adaptam são 8 e 16. Existem vantagens e desvantagens de se utilizar a notação octal ou a hexadecimal. É mais fácil para o leigo utilizar a notação octal porque seus dígitos são todos numéricos entre Ø e 7. Entretanto, a notação octal apresenta no presente caso a desvantagem seguinte: sendo a palavra de oito bits, é impossível dividí-la em partes iguais de três bits cada.

Assim, apesar de ser possível representar constantes de oito bits em octal, usando três dígitos, deve-se levar em conta que neste caso o dígito mais significativo representa dois bits apenas, e não três. Êste detalhe, apesar de ser irrelevante em sí mesmo, induz no presente caso um outro problema, o que permitiu que se decidisse pela não utilização da numeração octal em todo o "software" desenvolvido: ao se analisar o conjunto de instruções , nota-se que a grande maioria das instruções pode ter seus códigos divididos de 4 em 4 bits, sendo que cada um dêstes conjuntos de 4 bits tem significado próprio. Além disso, o código de operação de todas as instruções de referência à memória está contido em seus 4 primeiros bits, e nas instruções restantes os mesmos 4 bits representam o grupo de instruções a que pertence uma instrução particular.

Com essas considerações, e levando-se em conta que uma palavra é divisível em dois grupos iguais de 4 bits, chegou-se à conclusão que a notação hexadecimal é a melhor para a representação de números binários. Assim, pela simples leitura em hexadecimal do número binário correspondente a uma instrução qualquer pode-se identificá-la facilmente depois de pouco tempo de familiarização com os códigos.

A notação decimal é a mais natural entre todas as representações das constantes, sendo portanto muito conveniente que se

permita escrever constantes em decimal.

Pelo fato de que todos os periféricor utilizados na configuração utilizam o código ASCII para a representação de caracteres, este terceiro modo de declarar as constantes torna-se bastante importante. Se algum periférico utilizasse outros tipos de códigos para a representação dos caracteres, seria bom que se permitisse ainda a representação de constantes em tais códigos.Êste não é, entretanto, o caso.

Naturalmente, se fôr permitido o uso de tipos diversos de dados, será necessário identificá-los de algum modo. Assim , tem-se diversas possibilidades. Por exemplo, poder-se-ia associar a cada tipo de dado uma pseudo instrução específica, a qual admitirá unicamente operandos do tipo ao qual se refere, ou então utilizar uma única pseudo instrução, e fornecer uma indicação explícita sôbre o tipo de dado declarado no operando.Pode-se ainda utilizar um esquema misto, em que se tenha mais de uma pseudo instrução, sendo que cada uma delas pode ter operandos identificados quanto ao tipo.

Quando se tem a possibilidade de declarar dados de comprimento variável, é interessante que se utilize o esquema misto. Nêste caso, pode-se-ia ter, por exemplo, uma pseudo para cada tipo de dado, sendo declarado no operando o comprimento do dado na parte de identificação. Êste não é o caso, pois tem-se dados de um comprimento apenas: dados inteiros de oito bits.Decidiu-se utilizar, para todas estas possibilidades, uma única pseudo (DEFC) sendo que, a exemplo dos endereços absolutos, o operando deverá vir acompanhado de um identificador de tipo. Ter-se-ia portanto para o DEFC as seguintes possibilidades:

Tipo de Operando	Identificador	Exemplos
Decimal inteiro	nenhum	DEFC 53 DEFC -21
Hexadecimal	/	DEFC /2F DEFC -/35
ASCII	@	DEFC @ P DEFC -@ 1

2.5.6.4. A Pseudo COM

Quando se escreve um programa relocável composto de um número grande de segmentos e/ou subrotinas, independentemente desenvolvidos, a comunicação entre os diversos módulos do programa pode apresentar-se como um problema relativamente sério. Assim, se uma rotina utilizar M variáveis provenientes do programa que a chamou, e devolver a êste programa N outras variáveis, ter-se-á um total máximo de M+N variáveis envolvidas na transferência de informações. Esta transferência pode ser executada fornecendo-se para a subrotina, em posições convenientes, as próprias variáveis ou então seus endereços. Isto é feito normalmente por meio de uma "sequência de chamada", onde as palavras que seguem a chamada da rotina em questão não são código executável, mas informações sôbre os parâmetros. Tal procedimento acarreta a necessidade de se escrever (e portanto gerar código objeto) uma sequência de chamada a cada vez que se chama uma subrotina. Por outro lado, durante a execução da subrotina chamada, dever-se-á providenciar a transferência dos parâmetros, o cálculo dos resultados e a sua devolução ao programa que a chamou, devendo-se além disso acertar o endereço de retôrno para que não sejam executadas as palavras correspondentes às informações sôbre os parâmetros, na sequência de chamada.

Como no caso tal transferência de parâmetros é trabalhosa e demorada, consumindo, além disso, muita memória, é bastante conveniente que se possa dispor de um outro meio menos dispendioso para a comunicação entre os diversos módulos de que é composto o programa.

Um meio clássico relativamente fácil de implementar, para contornar êste problema é o uso de área comum de dados ("common"). A área comum começa em uma posição conhecida de memória, sendo acessível aos diversos módulos. Organizando-se convenientemente esta área, as diversas rotinas poderão comunicar-se sem a necessidade de transferência de parâmetros, reservando-se êste último recurso apenas para casos especiais, onde seja mais conveniente utilizá-lo. Para a definição e organização da área comum criou-se a pseudo COM, cuja sintaxe é idêntica à da pseudo BLOC (2.5.6.5).

O tratamento da pseudo COM se resume em deslocar um contador de memória comum utilizada, no primeiro passo. No final do passo 1 ter-se-á o total de área comum declarada na rotina, informação que será anexada às demais no bloco de NOME do programa objeto. Ter-se-á também construído a tabela de símbolos, com os endereços das variáveis que constam da área comum definidas em relação à origem da mesma, e portanto assinalados como sendo pertencentes à área comum. No segundo passo, as pseudos COM não terão utilidade, sendo ignoradas. A atribuição de endereços efetivos para a área comum é feita pelo ligador-relocador (ref. 2).

2.5.6.5. A Pseudo BLOC

Tabelas são recursos bastante utilizados em programação pela facilidade que podem trazer à resolução de um grande número de problemas, principalmente quando se dispõe de indexadores em "hardware". Às vêzes as tabelas são utilizadas apenas para consulta, como foi visto no caso das tabelas de mnemônicos no reconhecimento dos comandos. Em outros casos, também frequentes, as tabelas são construídas na ocasião da execução do programa, como no caso das tabelas de símbolos. Neste último exemplo, sente-se a necessidade de algo que, na fase de montagem do programa, reserve a área necessária para a construção de tais tabelas na fase de execução do mesmo. Existe a possibilidade de reservar tal área de várias maneiras utilizando os recursos já vistos. Por exemplo, poder-se-ia definir uma nova origem para o código que seria montado a seguir, sendo a área intermediária reservada para a tabela. Outra maneira seria preencher a área desejada com instruções ou então com constantes, sem função lógica. Tal procedimento, apesar de ser válido e funcionar bem, apresenta o incoveniente de não ser mnemônico. Para melhorar esta característica do montador, acrescentou-se a pseudo instrução BLOC,que serve para reservar blocos de memória. Seu operando deverá ser uma constante.

O tratamento desta pseudo é trivial, limitando-se, no primeiro passo, a acertar o ponteiro para a próxima posição de

memória a ser preenchida, e, no segundo passo, além de executar este acerto, descarregar o código objeto que já estava acumulado e iniciar um outro bloco de código com novo endereço inicial.

A diferença entre as pseudos BLOC e COM reside no fato de a BLOC reservar área na região de código (relocável em relação ao início do programa) ao passo que a pseudo COM reserva área na região comum (relocável em relação ao início da área comum). Em ambos os casos, se houver rótulo na pseudo instrução, tal rótulo representará o endereço da primeira palavra reservada pelo bloco definido pela mesma.

2.5.6.6. A Pseudo EQU

Apesar de não se ter possibilidade de declarar um número muito grande de símbolos no mesmo programa por falta de espaço na tabela de símbolos, é muito frequente que, depois de escrito o programa, se venha a detetar a definição supérflua de posições de memória para utilização temporária e de uso restrito. Para permitir um aproveitamento melhor da memória no instante da execução, é conveniente que se possa atribuir à mesma posição de memória vários nomes, afim de que o programa, já escrito, não tenha de ser perfurado novamente ou editado por extenso com a finalidade de modificar os nomes das variáveis em questão. Assim, pela supressão da definição das posições de memória desnecessárias e sua substituição por comandos de equivalência de símbolos pode-se preservar os nomes mnemônicos de tais símbolos e, ao mesmo tempo, poupar memória no programa objeto. Além disto, pode-se desejar atribuir a uma posição absoluta de memória ou então a um determinado elemento de um bloco um nome mnemônico. Tais ações podem ser executadas pela pseudo EQU, embora possam algumas delas ser executadas também por combinações das outras pseudos já vistas.

Sintaticamente, a pseudo EQU apresenta um rótulo, que deve ser obrigatório, uma vez que é ele quem determina o nome da posição de memória em questão, e o operando propriamente dito, o qual poderá ser uma referência à memória que já tenha si

do definida anteriormente, e que não seja global externo.

A restrição imposta de que o símbolo deva ter definição prévia não tira a generalidade, uma vez que, se em último caso as pseudos EQU forem os últimos comandos do programa, todas as variáveis bem como todas as referências que poderiam interessar ao programador, já estariam definidas.

Se tal restrição não fosse imposta, bem como a de não ser externa a referência em questão, haveria necessidade de criar duas novas tabelas, uma para o tratamento das equivalências entre símbolos e outra para o das equivalências com posições relativas a símbolos globais externos, o que no caso não é recomendável pois a falta de memória para as tabelas e para as rotinas de tratamento não justifica a pequena flexibilidade adicional que tal procedimento traria.

Com tais imposições e restrições, a implantação da pseudo EQU torna-se também simples:

Calculado o endereço do operando, e verificada a não definição anterior do rótulo, basta atribuir ao rótulo tal endereço, na tabela de símbolos, no primeiro passo. No segundo passo, EQU será ignorada uma vez que sua única função já foi executada no passo 1.

Caso houvesse sido implementado o tratamento de equivalências entre um rótulo e uma posição de memória relativa a um símbolo externo, ter-se-ia tido a necessidade de acrescentar ao código objeto relocável correspondente à referência externa uma nova informação, a do deslocamento em relação a tal posição.Esta informação seria então manipulada, na ocasião da relocação do programa, pelo ligador-relocador (ref. 2), e deveria acompanhar todas as referências externas. Além disto, no segundo passo do montador, deveria ser gerado um bloco especial de equivalência, contendo a informação para o ligador-relocador sobre o nome do símbolo referenciado e sua relação com o verdadeiro símbolo global externo.

Tal símbolo poderia constar, no código objeto, como se fosse um símbolo externo e ao mesmo tempo uma variável global nova. Sua relação com os demais símbolos apareceria num bloco de equivalência, para posterior resolução pelo ligador-relocador.

2.5.6.7. O Controle BLT e D

Com o intuito de permitir ao usuário um comando sobre as diversas saídas que são fornecidas pelo montador, pode-se, por exemplo, permitir que tais listagens sejam opcionais, sendo a opção especificada através das chaves do painel. Tal procedimento não é muito conveniente principalmente quando se tem em vista que nem sempre é o próprio usuário quem opera o computador, bem como que o comando pelo painel não é muito recomendável quando se dispõe de um sistema operacional. Assim, o controle de tais opções poderia ser feito por meio de controles adequados para listagens, tabelas de símbolos, ou geração de fita objeto. Tais opções foram colocadas todas em um comando (BLT), que, quando não for declarada, mantém ligadas as opções de fita binária, listagem e tabela de símbolos, e quando explícita, cada letra (B,L ou T) presente liga a opção correspondente:(ref. 10).

B - Fita Binária

L - Listagem

T - Tabela de Símbolos

O tratamento deste comando é imediato, limitando-se a guardar a informação das opções em posições testáveis pelas rotinas de geração correspondentes.

Uma nova informação (Dn), onde n é um dígito hexadecimal, pode ser declarada neste controle. Seu objetivo é informar ao ligador-relocador que o programa em questão é uma rotina de entrada e saída com interrupção ("driver") para o dispositivo cujo endereço de entrada e saída é n. O tratamento desta informação está detalhado na ref. 2.

2.5.6.8. As Pseudos DEFE e DEFI

Com a possibilidade de se referenciar indiretamente uma posição de memória, surge a necessidade de gerar, em posições conhecidas de memória, os endereços que servirão como ligação entre

a instrução a executar e o operando, definido o endereço deste. Para facilitar a geração de tais endereços criou-se a pseudo DEFE que cria em duas posições sucessivas de memória uma constante de 16 bits cujos 12 bits menos significativos são o endereço da posição de memória referenciada como operando. Os 4 bits mais significativos são feitos iguais a zero.

Devido à possibilidade de endereçar indiretamente em mais de um nível, é necessário poder-se definir, nas posições de memória que definem endereço, que este ainda não é endereço do operando, mas um apontador para outro endereço. Para isto é utilizado o "menos significativos" dos quatro bits mais significativos, o qual indica, se "zero", endereçamento direto, e se "um", endereçamento indireto. Para gerar endereços indiretos, utiliza-se a pseudo instrução DEFI.

2.5.6.9. A Pseudo FIM

Toda linguagem de computador precisa informar ao respectivo tradutor até onde vai o programa fonte. Há inúmeras maneiras de executar tal tarefa, sendo que a mais comum é a de se utilizar um mnemônico adequado. Com esta finalidade criou-se a pseudo FIM, a qual além de servir como final físico do programa fonte, pode assumir, no caso de um programa principal, um operando que informa ao ligador-relocador qual é a primeira instrução a ser executada. Assim, nos programas principais relocáveis, a pseudo FIM deverá ter um operando, que é referência geral à memória, e que indicará o endereço de execução do mesmo.

O tratamento desta pseudo é o seguinte:

No passo 1, lida a pseudo FIM, é feito um teste de consistência da tabela de símbolos, sendo esta impressa ou não de acordo com a opção estabelecida pela comando BLT. Se houver símbolos indefinidos, estes serão listados e o passo 2 não será executado. Isto também acontecerá se algum erro de sintaxe for detetado em algum ponto do programa.

Caso não tenha ocorrido nenhuma anormalidade, o passo 2 será executado, e quando for detetada a pseudo FIM no passo 2, haverá o descarregamento do restante do código objeto ainda exis

tente na memória (se isto houver sido especificado no controle BLT), e a execução do montador será reiniciada.

2.6. O Programa Principal

De posse de todas as definições das características sintáticas e semânticas da linguagem do montador, bem como dos algoritmos a utilizar em cada caso, passou-se a elaborar o programa principal. Como foi visto em 2.3, o montador terá dois passos, tendo o primeiro passo, como finalidade principal, a montagem e listagem da tabela de símbolos e a deteção de erros de sintaxe. Não havendo símbolos indefinidos, nem erros de sintaxe, o passo 2 será executado, produzindo a listagem do programa e a fita com o programa objeto. Descreve-se a seguir a implementação de cada - uma destas etapas.

2.6.1. O Primeiro Passo

Posicionados alguns contadores, indicadores e apontadores, passa-se a ler e analisar o programa fonte, linha a linha . Numa versão com memória de massa, uma imagem do programa fonte vai sendo guardada para uso no 2º passo. Como foi estabelecido,o primeiro comando de um programa deverá identificá-lo como absoluto ou relocável. Se o programa for absoluto, seu primeiro comando deverá ser a pseudo ORG, caso contrário NOME, SUBR, ou SEGM. Se esta condição não for satisfeita, será fornecida uma mensagem de erro, e então o passo 1 deverá ser reiniciado após as devidas correções de programa fonte. A rotina de erro contabiliza o número de erros detetados no programa para futura consulta no final do passo 1.

A seguir, cada comando lido será tratado como descrito a seguir (fig. 2.6.1.1).

Manipula-se primeiramente o campo dos rótulos, analisando-se sua validade. Se o rótulo for válido, procura-se o mesmo na tabela de símbolos, colocando-o e definindo-o, ou definindo-o apenas se já estiver presente mas indefinido. Caso o rótulo

Fig. 2.4.1.1 - Diagrama de blocos simplificado do primeiro passo do montador

já se encontre definido na tabela, será fornecida uma mensagem de erro de dupla definição. Se o rótulo não for válido, de acordo com as regras a que se chegou em 2.3, será fornecida uma mensagem de erro de rótulo ilegal.

Em seguida, é analisado o campo dos mnemônicos, sendo fornecida uma mensagem de erro no caso de não existir tal mnemônico. Constatada a existência do mnemônico, é chamada a subrotina de identificação descrita em 2.5.4. Novamente uma mensagem de erro será impressa no caso de o mnemônico utilizado não ser válido. Se o mnemônico em questão for encontrado na tabela dos mnemônicos, ter-se-á, ao fim da execução desta subrotina, diversas informações relativas ao mesmo, entre as quais a do grupo a que ele pertence, da necessidade ou não de operando, e se este deve ser numérico ou não. Assim, utilizando tais informaçẽs, passa-se à análise do operando, se este for exigido pelo comando. Se o comando pedir operando do tipo numérico, o campo dos operandos é devidamente analisado, sendo fornecida mensagem de erro caso seja ilegal. Isto pode ser executado de várias maneiras, sendo que se utilizou, no presente caso, a própria rotina de conversão para detetar os possíveis erros de construção da constante em questão. Naturalmente, no passo 1 não haveria necessidade de se chegar a converter a constante, bastando apenas o teste de validade. Entretanto, tal procedimento se justifica tendo-se em vista que o programa de teste é bastante extenso, havendo portanto, caso seja utilizado, uma perda substancial de memória, uma vez que a rotina de conversão também deverá estar na memória por ser utilizada em outras ocasiões. Portanto, no presente caso, a própria rotina de conversão incorpora os testes de validade da constante.

Se o operando do comando em questão puder ser não numérico, deverá ser executada uma análise com a finalidade de encontrar o tipo de operando com que se está trabalhando. Descoberto o tipo, desvia-se para rotinas de teste que verificam a validade dos mesmos. No caso de o operando em questão ser do tipo simbólico, relativo ou não, deve-se procurar o símbolo em questão na tabela de símbolos, colocando-o no final da mesma e indefinindo-o caso se trate de um símbolo ainda não referenciado, e ignorando-o caso já esteja presente na tabela. Obvia

mente não é dado este tratamento ao operando da pseudo EXT, o qual deve ser colocado e definido como global externo na tabela de símbolos, tendo portanto manipulação semelhante à dos rótulos. Como será visto, este procedimento, repetido para cada instrução, compõe a tabela de símbolos do programa fonte em análise.

Para que a tabela seja construída adequadamente falta um detalhe: quando se define o endereço correspondente a um símbolo, isto é, quando o símbolo aparece como rótulo, o que se faz normalmente é guardar, na posição correspondente na tabela, o endereço de memória em que será gerado o código objeto relativo ao comando analisado.

Além disto, a cada comando analisado, deve-se atualizar um contador, (CI) que indica a próxima posição de memória a ser preenchida.

Isto no caso das instruções, é implementado somando-se 1 ou 2 ao valor do contador conforme a instrução seja curta ou longa, respectivamente, informação esta que é fornecida pela propria rotina de pesquisa de tabela de mnemônicos. No caso das pseudos, o tratamento dado ao contador depende da pseudo, e é executado na particular rotina de tratamento correspondente à mesma. O processo descrito continua até que seja detetada a pseudo FIM, quando é analisada a consistência da tabela de símbolos, e testado o contador de erros. A Tabela de símbolos será então listada se isto houver sido especificado.

Se houver algum símbolo indefinido ou se algum erro tiver sido detetado no programa (isto é verificado consultando-se um contador de erros), a execução do segundo passo será inibida, caso contrário o controle poderá ser passado imediatamente ao passo 2, uma vez que a tabela de símbolos já está devidamente construída e pronta para utilização.

2.6.2. O Segundo Passo

De posse da tabela de símbolos gerado no passo 1, passa-se à execução do segundo passo do montador, cujo objetivo

primordial é o de traduzir para linguagem de máquina o programa fonte. Para isso são utilizadas informações provenientes da tabela de símbolos e do próprio programa fonte.

Atribuídos valores iniciais às diversas variáveis do montador, o programa fonte deverá ser relido, seja a partir do próprio texto fonte em fita de papel, seja a partir de uma cópia do mesmo, produzida durante o primeiro passo na memória de massa do computador. Para cada linha do programa fonte, dá-se o tratamento descrito a seguir (fig. 2.6.2.1.).

Primeiramente é desprezado o campo de rótulos, passando-se imediatamente à identificação do mnemônico, como foi feito no primeiro passo. A seguir, é chamada uma subrotina que fornece, a partir das informações obtidas da rotina de identificação, a parte do código objeto da instrução que independe do operando (no caso de se tratar de uma instrução que exija operando). Caso o comando analisado seja uma pseudo, o processamento é desviado para a respectiva rotina de tratamento.

Em seguida, caso a instrução analisada não exija operando, a geração estará completa, sendo o código objeto entregue à rotina de saída. Se a instrução exigir operando, entretanto, é necessário que as informações provenientes do mesmo sejam, antes disso, incorporadas ao código objeto. Tais informações são calculadas a partir do operando e da tabela de símbolos, por meio de uma rotina geral de conversão que incorpora - também uma busca de informações de endereçamento na tabela de símbolos.

A rotina de saída do código objeto tem como função gerar o código em um formato compatível com o programa carregador ou com o ligador-relocador, além de transportar todas essas informações para o meio externo. Assim, uma área de rascunho, na memória, é utilizada para guardar o código objeto à medida que este for sendo gerado. No instante em que esta área de rascunho é totalmente preenchida, a rotina de saída transfere toda esta informação para o meio externo, em formato compatível com os programas carregadores, caso a fita binária tenha sido solicitada pelo programador.

Fig. 2.6.3.1 - Diagrama de blocos simplificado da lógica do segundo passo do montador

Depois de fornecido o código objeto montado à rotina de saída, é verificado se se deseja a listagem do programa. Caso isto se verifique, são impressos numa unidade de saída o número da linha, o valor de CI (endereço da instrução na memória na ocasião de execução) e o código objeto, em formato hexadecimal, ao lado das informações de relocação do código objeto e da listagem formada do programa fonte.

O procedimento descrito é repetido para todas as linhas do programa fonte, até que uma pseudo FIM seja detetada. Nesta ocasião, é forçada a saída do último bloco do código objeto, e o passo 2 é encerrado.

2.6.3. Observações sobre a Ampliação dos recursos do Montador

Com os dois passo, descritos acima, tem-se a possibilidade de geração de código objeto apenas a partir de programas escritos na linguagem fonte descrita em 2.3 e 2.4, a qual está intimamente relacionada com a linguagem de máquina do computador. Assim, como a máquina não reconhece constantes em ponto flutuante, esta linguagem não permite a declaração e a manipulação de tais constantes. Analogamente, funções mais complexas que as instruções de máquina não são reconhecidas pelos dois passos do montador.

Se se desejar dar ao usuário a possibilidade de definir seus próprios mnemônicos em função dos já existentes, e permitir a declaração e manipulação das constantes em ponto flutuante, poder-se-ia ampliar as capacidades dos passos para que consigam reconhecer os mnemônicos de definição e manipulação de constantes em ponto flutuante como macros implícitas, e, a partir de sua expansão em comandos elementares, gerar corretamente o código desejado.

No presente caso, pela necessidade de uma área excessiva da memória que as novas rotinas de conversão ponto flutuante-binário exigem, ao lado das rotinas de expansão das macros implícitas, tal procedimento é inviável além de ser bastante inflexí-

vel, se implementado desta maneira.

Outra alternativa, mais trabalhosa do ponto de vista de utilização, porém mais versátil, é a de escrever um novo programa, independente dos dois passos do montador, que funcionaria como passo Zero, e cuja finalidade seria a de expandir todas as macros implícitas já definidas, bem como permitir a definição e a expansão de macros criadas pelo usuário. Este programa teria como finalidade gerar, ao final de sua execução, um programa fonte compatível com os dois passos, já existentes, do montador, podendo ser utilizado futuramente para o tratamento de uma linguagem intermediária entre a linguagem do montador e uma linguagem de alto nível que venha a ser implementada. A utilização deste expansor seria necessária apenas quando da utilização, por parte do usuário, de macros implícitas ou por ele definidas, sendo portanto dispensável se o programa utilizar apenas os recursos da máquina, como é o caso mais frequente. Um programa deste tipo está sendo desenvolvido atualmente para o Patinho Feio, e deverá fazer parte do grupo de novos programas a serem implantados em um pequeno sistema operacional para disco, ora em fase de projeto.

A operação do montador no computador sem memória de massa e sem recursos de saída rápida de informações é bastante demorada e portanto requer, para a obtenção de resultados mais rápidos, que se lance mão de outros meios, principalmente quando se trata de programas longos, como é o caso dos aqui descritos.

Com esta finalidade, foi desenvolvido, para o sistema Hewlett-Packard HP-2116-B, um programa interpretador das instruções do minicomputador utilizado, o qual permite a utilização dos periféricos rápidos e da memória de disco e de fita magnética do sistema HP-2116-B em substituição aos terminais lentos do Patinho Feio. Assim, embora no interpretador o tempo de processamento seja dezenas de vezes maior, o tempo da saída impressa ou perfurada é centenas de vezes menor, e como os programas são limitados em velocidade pelo grande volume de entrada e saída que executam, há uma vantagem muito grande de testá-los no interpretador, pois este fornece, além dos resultados do programa, facilidades de "dumps" de memória, "trace", etc., sem a necessidade de utilização da memória do próprio Patinho Feio para guardar os pro

gramas geradores de tais listagens.

No capítulo 3 está descrito um interpretador desenvolvido para esta finalidade.

Com as facilidades do interpretador torna-se possível o desenvolvimento da versão dos programas aqui descritos para uma configuração do Patinho Feio com memória de massa, sendo por tanto utilizável no desenvolvimento de um sistema operacional, bem como na adaptação dos demais programas para a geração de códigos em linguagens intermediárias diretamente no disco, antes que este esteja definitivamente implantado.

Recebendo uma ordem para a tradução de um programa, tal sistema operacional deveria encarregar-se de trazer da memória de massa primeiramente o "passo zero", e executá-lo e gravar no disco a imagem do programa fonte expandido para o passo 1. Em seguida seria trazido do disco o passo 1, que processaria o programa fonte lendo-o do disco e gerando na memória a tabela de símbolos. Em seguida o passo 2 seria trazido, para ser executado, gerando, a partir do programa fonte e da tabela de símbolos gerada no passo 1, o programa objeto, guardando-o no disco, para uso do ligador-relocador. Finalmente este seria trazido do disco e, utilizando o programa objeto gerado e rotinas de biblioteca, também armazenadas em disco, poderia gerar, ainda no disco, o programa na sua versão absoluta, pronta para ser executada. Um comando de execução para o sistema operacional traria do disco para a memória o programa montado, e iniciaria sua execução. Naturalmente desta maneira seria possível optar pela saída do programa nas diversas linguagens intermediárias para um meio externo. Como se vê, embora no próprio computador esta tarefa seja trabalhosa e demorada sem o auxílio de um disco, no interpretador ela poderá ser automatizada tornando viável o projeto e desenvolvimento de sistemas maiores como compiladores de linguagem do alto nível, interpretadores, etc. em um tempo bastante menor que o necessário para tal desenvolvimento no próprio computador. Assim, implementados os novos programas, estes podem ir sendo utilizados no interpretador até que se tenha funcionando no Patinho Feio um disco ou fita magnética, ocasião em que um simples transporte de todo o "software" desenvolvido permitirá a utilização imediata do mesmo no próprio Patinho Feio.

As modificações necessárias nos dois passos do montador para o funcionamento adequado em um sistema operacional são relativamente simples, devendo-se adicionar uma rotina de entrada e saída para o disco, e um teste da opção de saída simultânea em fita perfurada, e, naturalmente, chamadas de supervisor convenientes para que seja feito automaticamente o transporte dos diversos segmentos do programa para uma área de sobreposição ("overlay"). Naturalmente, haverá também a necessidade de se estruturar corretamente os programas que deverão ser executados sob a supervisão do sistema operacional, criando-se para cada um deles um segmento residente, uma área da sobreposição, e uma área comum de dados, o que também não é muito complicado para programas bem projetados.

Um outro recurso, que se torna cada vez mais importante à medida que o tamanho do programa aumenta, é o de geração de tabelas de símbolos especiais, denominadas tabelas de referências cruzadas "cross-reference symbol tables". Trata-se de tabelas onde todos os símbolos definidos no programa fonte são listados em ordem alfabética ao lado do número da linha em que foram definidos, e do conjunto dos números das linhas em que foram referenciados. Sua utilidade principal é a de documentar o programa, permitindo que os símbolos procurados sejam rapidamente localizados, o que facilita o trabalho do programador nas fases de depuração do programa, e de ampliação ou modificação de um programa já depurado.

Uma versão do montador ("cross-assembler"), incluindo todas estas facilidades e opções foi escrita na linguagem ALGOL para o sistema HP-2116-B, e se mostra bastante cômoda no desenvolvimento de vários dos módulos de software desenvolvidos para o Patinho Feio, principalmente quando conjugado com o uso do simulador-interpretador, pela facilidade de utilização dos recursos do sistema operacional do computador HP-2116-B. Nesta versão, estão incluídas duas facilidades a mais: a listagem da tabela de símbolos em ordem alfabética e a geração opcional da tabela de referências cruzadas. Esta última foi incorporada ao controle BLT. Se neste controle for incluído um símbolo "C", o montador fornecerá a tabela de referências cru

zadas do programa.Além disso, uma das chaves do painel permite que se opte por uma tabela de utilização de cada mnemônico da linguagem, para efeito de estatísticas. Apesar de ser estruturado exatamente como o da versão para o Patinho Feio, este montador tem a vantagem de utilizar arquivos em disco do sistema HP-2116-B, com todas as facilidades de edição "on line", o que dispensa a geração de várias fitas durante a tentativa de eliminação de erros de programação. Além disso, esta versão permite uma operação mais rápida, uma vez que, se a entrada for feita via fita de papel, esta deverá ser lida uma vez apenas . Quanto à deteção de erros de linguagem, pode-se dizer que as duas versões são equivalentes, embora nesta versão para o HP-2116-B as mensagens de erro sejam fornecidas por extenso, o que dispensa o uso de tabelas de erros pelo usuário.

Para o desenvolvimento deste "cross-assembler" foi escolhido o sistema HP-2116-B por sua compatibilidade de entrada e saída com o Patinho Feio (fita de papel), sem o que,tal "cross-assembler" não seria tão útil.

2.6.4. Críticas

Tratando-se de um programa escrito de maneira bastante compacta com a finalidade de economizar memória, o montador tem a característica de exigir muito processamento, o que o tornará relativamente lento quando de sua adaptação a um sistema operacional com entrada e saída em disco, pois, sendo grande o volume de entrada e saída, e sendo esta, no caso de disco, bastante rápida, verificar-se-á que os tempos de processamento e de entrada e saída serão de ordens de grandeza próximas. No entanto, se se considerar as entradas e saídas normais, constatar-se-á que o tempo de processamento, neste caso é desprezível. Para o caso do disco, porém, talvez seja interessante reescrever o programa, eliminando processamentos desnecessários, escrevendo-o portanto mais por extenso e segmentando-o convenientemente, deixando na memória as rotinas mais usadas, as quais deverão ser otimizadas em relação ao tempo de execução. Minimizando os acessos ao disco devidos à segmentação, poder-se-á conseguir tempos melhores tornando a operação

mais eficiente.

Sendo residente na memória, a tabela de símbolos tem um tamanho máximo que depende do espaço ocupado pelo restante do programa, e pela respectiva subrotina de manipulação. Por razões de economia de memória, esta subrotina foi escrita da maneira mais compacta possível, como se viu em 2.5.3. Isto levou a um limite superior do número de símbolos igual a 256, devido ao fato de o registrador de índice ter oito bits apenas. Assim, se se desejar ampliar a capacidade de tabela de símbolos, deve-se abandonar o endereçamento indexado simples e passar a esquemas de endereçamento indireto, ou de modificação de instruções, os quais envolvem aritmética de dupla precisão tornando mais extensa e mais demorada a pesquisa. Se se desejar definir mais de 256 símbolos no programa, será necessário utilizar esta técnica ou então a segmentação da tabela de símbolos, mantendo uma parte residente na memória e outra em um meio externo. Este último recurso só é prático no caso em que se disponha de memória de massa, caso em que será possível a definição de um número muito grande de símbolos. Para uma configuração sem disco ou fita magnética, resta o recurso de ampliar a tabela residente e modificar a rotina de pesquisa.

Como os rótulos definidos são transformados sempre em símbolos de três caracteres, é possível que o usuário tenha referenciado mais de um rótulo, diferentes entre sí, mas cujos transformados sejam iguais, tendo definido apenas um deles. Para o montador, tudo se passa como se ambos fossem o mesmo rótulo, não sendo portanto possível a deteção do erro. Situações como esta acontecem frequentemente quando se acrescentam novos trechos a um programa extenso em teste, sendo o único meio prático de evitar que tal aconteça o exame cuidadoso da tabela de referências cruzadas antes da modificação do programa. Para a versão com disco, é possível mudar o critério de compactação dos símbolos, tornando significativos mais caracteres e, portanto, diminuindo a possibilidade de ocorrência de fatos como este.

Como é de se esperar, esta modificação exige uma alteração substancial de um grande número de rotinas do montador, pois toda a estrutura das rotinas que utilizam tabelas baseia-se nesta característica.

CAPÍTULO 3. UM SIMULADOR-INTERPRETADOR PARA A LINGUAGEM DE MÁQUINA DO PATINHO FEIO

3. UM SIMULADOR-INTERPRETADOR PARA A LINGUAGEM DE MÁQUINA DO PATINHO FEIO.

Para computadores de pequeno porte, são grandes as vantagens de se ter em um outro computador disponível, com maiores recursos, um programa que, quando executado, faça com que este se comporte o mais próximo possível como se fosse o primeiro. Sempre que se desejar construir um programa com objetivo semelhante a este, uma decisão básica deverá ser tomada: até que ponto o sistema simulador (computador hospedeiro + programa de simulação) deverá exibir o mesmo comportamento do sistema simulado.

3.1 Definição das especificações do programa de simulação.

É possível construir programas de simulação em diversos níveis, dependendo do grau de detalhe que se deseja analisar nos resultados. (ref. 5).

Entre os diversos níveis de simulação, deve-se escolher aquele que melhor se adapte à finalidade a que se propõe o simulador. Assim, no exemplo implementado, todas as instruções de máquina não relacionadas com entrada e saída foram simuladas em nível de registrador, sendo observáveis apenas os resultados da execução da instrução e não as suas causas, como deve ser o caso em um simulador de arquitetura do tipo do que foi desenvolvido no sistema IBM 1130 na época do projeto lógico do Patinho Feio, no qual é possível a observação, passo a passo, das diversas fases da execução de cada instrução. Naturalmente este tipo de simulação é muito útil na fase de depuração do "hardware", tendo porém a desvantagem de ser bastante lento, devido ao grande número de detalhes considerados, não sendo portanto indicado no presente caso, onde o objetivo é apenas a simulação da execução de programas.

O tratamento das instruções de entrada e saída pode ser considerado o mais trabalhoso do programa. Para um teste preliminar do simulador, foi implementada uma versão onde o sistema de interrupção do computador simulado não foi levado em conta, sendo consideradas, das instruções de entrada e saída, apenas as indispensáveis. Nesta versão as instruções de "salto se estado estiver ligado" foram substituídas por um salto incondicional, as instruções de "entrada" e de "saída" de dados, simuladas mediante trans

ferência direta de dados entre a unidade central de processamento
e os periféricos do computador hospedeiro.As demais instruções de
entrada e saída foram ignoradas. Com isto, foi possível fazer um
teste de todo o conjunto de instruções que não manipula entrada e
saída, tendo sido possível ainda utilizar o simulador (no caso,
apenas um interpretador) para a execução de qualquer programa que
não utilizasse o sistema de interrupção do Patinho Feio.

Com esta restrição, o uso de tal interpretador será sufi
ciente apenas para os programas particulares que executam entrada
e saída por meio da técnica "wait for flag" (ref.6) não sendo pos
sível no mesmo a simulação de programas que utilizam a técnica de
interrupção.

Devido à grande conveniência de se poder depurar um pro
grama de tratamento de interrupção por meio do seu acompanhamento
passo a passo ("trace"), elaborou-se uma segunda versão do simula
dor, na qual as instruções normais continuaram sendo interpreta
das como na primeira versão, mas onde todas as instruções de en
trada e saída passaram a ser consideradas e totalmente simuladas
no nível da lógica das interfaces, com o auxílio de um modelo sim
plificado dos circuitos das mesmas. A introdução de um parâmetro
de tempo também foi necessária, como será visto adiante. Com esta
segunda versão tornou-se possível a execução de qualquer programa
do Patinho Feio no simulador-interpretador construído.

Havendo no computador hospedeiro recursos e facilidades
não disponíveis no computador simulado, é interessante que se pos
sa escolher os que se deseja utilizar, em cada caso. Para isto,
foi desenvolvida uma rotina interpretadora de comandos de console,
os quais permitem ao operador configurar o simulador às conveniên
cias de cada programa. Os comandos de console dotam o simulador de
alguns recursos adicionais em relação ao próprio computador, como,
por exemplo, o carregamento automático de um "bootstrap" (progra
ma que serve para carregar outros programas), a opção de "traces"
(listagem do andamento do programa, instrução por instrução, para
efeito de acompanhamento) e "dumps" (listagens,em hexadecimal, do
conteúdo da memória), a possibilidade de simplificar a simulação da
interrupção para acelerar a execução de rotinas de entrada e saí
da, e assim por diante. Além disto, algumas chaves do painel foram
escolhidas para representar o registrador de chaves do Patinho Feio,

e as demais foram utilizadas para controlar algumas opções do simulador, como por exemplo, ligar e desligar "traces", simular botões do painel, passar o controle para a console, etc.

Partindo dessas considerações e levando-se em conta o objetivo a que se propõe o simulador, pode-se listar algumas características desejáveis:

a) Fácil utilização - Para isto, os comandos de console e as opções feitas pelas chaves do painel devem constituir um conjunto suficientemente poderoso para que a operação do simulador não se torne mais complexa que a do próprio computador simulado.

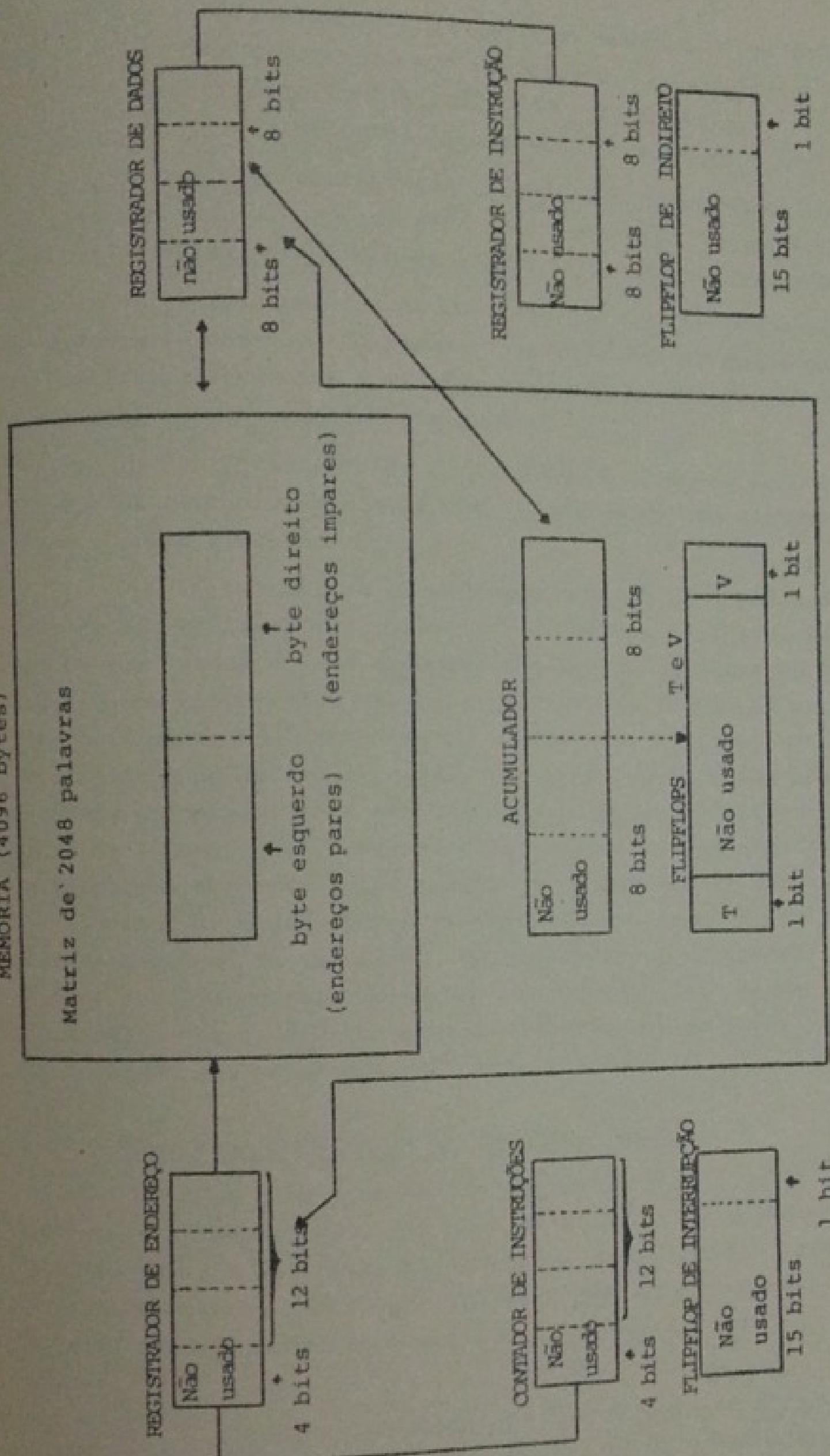
b) Deve ser tão rápido quanto possível-Pela própria característica de interpretador, a velocidade deste programa de simulação é baixa. No entanto, pode ser acelerada pela introdução de simplificações no tratamento das diversas instruções.

Assim, a execução das instruções normais pode ser simulada em nível de registradores, sem que detalhes desnecessários normalmente incluídos em simuladores de arquitetura sejam considerados. Desta maneira, a velocidade da interpretação de um programa é grandemente aumentada. Para aumentar ainda mais esta velocidade, os programas de simulação devem ser bastante otimizados, devendo ser escritos de preferência em linguagem "assembler" da máquina hospedeira.

c) Deve representar fielmente os resultados observáveis no computador simulado. Escolhidas as entidades de "hardware" a serem simuladas, todas as instruções deverão manipulá-las corretamente para que se possa, a qualquer altura do programa, extrair de cada uma delas a informação desejada.

d) Deve permitir a utilização da maioria dos recursos - existentes no computador hospedeiro. Como a conveniência do uso de um ou outro periférico depende do programa específico que se está executando, deve-se tornar o simulador apto a receber comandos, pela console, que associem a cada periférico do computador simulado um dispositivo conveniente existente no sistema hospedeiro.

e) Não deve restringir o uso de recursos do computador simulado. Para que qualquer programa possa ser executado indistintamente no computador ou no simulador, é conveniente que este esteja apto a simular todos os recursos normalmente utilizados no com-

putador simulado, afim de que não seja necessária a modificação do simulador sempre que surja uma situação ainda não considerada. Naturalmente, em alguns casos será impossível a simulação exata de todos estes recursos. Nestes casos, o simulador deverá permitir a substituição do recurso original utilizado pela máquina por algum outro similar , disponível no sistema hospedeiro.

f) O mecanismo de interrupção do computador simulado deverá ser representado, no simulador, de tal modo que qualquer programa de entrada e saída seja executável. Esta imposição faz com que o simulador deixe de ser apenas um interpretador, e exige que se construa ao menos um modelo que represente, sem detalhes supérfluos, todo o sistema de entrada e saída do computador simulado.

3.2 O interpretador das instruções

A rotina interpretadora das instruções que não manipulam entrada e saída é, embora extensa, bastante trivial. Consiste apenas na alteração do valor das variáveis que representam registradores ou posições de memória do computador simulado, de acordo com a instrução interpretada. Simulada a busca da instrução, o programa principal do simulador executa sua decodificação, e em seguida desvia para a rotina de execução correspondente à instrução. Quando se tratar de instruções de referência à memória, é feito um cálculo de endereço efetivo levando em conta o tipo específico de endereçamento utilizado na instrução (direto, indireto e/ou indexado). Em seguida é efetuada a ação correspondente à execução da instrução.

São representados, no modelo utilizado para o interpretador implementado no sistema hospedeiro HP-2116-B, os seguintes elementos do Patinho Feio (Fig.3.2.1):

- memória principal - matriz de 2048 palavras representando as 4096 palavras de 8 bits do Patinho Feio.
- acumulador - uma palavra
- índice e extensão - são as posições 0 e 1 da memória do Patinho Feio, representadas na primeira posição da matriz que simula a memória principal.
- "flipflops" T e V - uma palavra
- contador de instruções - uma palavra
- registrador de endereço da memória - uma palavra

Fig. 3.2.1 - O modelo dos componentes do Patinho Feio, utilizado na simulação das instruções em nível de registradores.

- registrador de dados da memória - uma palavra
- registrador de instruções - uma palavra
-"flipflop"de Indireto - uma palavra
-"flipflop"de Interrupção do sistema - uma palavra.

Como rotinas auxiliares, são empregadas, no manuseio da memória, uma rotina para ler o conteúdo de uma posição de memória, e outra para escrever um dado em determinado endereço.

São utilizados por esta rotina, os registradores de dados e de endereço da memória, simulados como acima descrito. Os demais registradores simulados são modificados pelas rotinas que simulam a execução das instruções.

O"flipflop"de interrupção é alterado pela instrução PUL ou pela ocorrência de uma "interrupção" no sistema de entrada e saída simulado.

As rotinas de execução constam de seqüênciasde instruções do HP-2116-B que equivalem, no modelo, à instrução simulada. Os resultados de sua execução são sempre armazenados nas variáveis que representam os registradores, os"flipflops"ou posições de memória do Patinho Feio. Todas estas rotinas, bem como as de simulação de entrada e saída, retornam para o mesmo ponto do programa principal, onde são testados a opção de "trace", os "pedidos de interrupção", etc. As diversas instruções simuladas nesta secção do programa estão descritas em detalhe na ref.1. Como já foi dito, no modelo foram representados apenas os efeitos globais da execução da instrução, desprezando-se, na maioria dos casos, os detalhes particulares de cada um. Com isto ganha-se em velocidade, o que é um fato bastante importante, em vista da natural lentidão dos processos de interpretação.

3.3 A Simulação do Sistema de Entrada e Saída.

A simulação do sistema de entrada e saída do Patinho Feio foi desenvolvida com base em um modelo simplificado do circuito real, modelo este que preserva as principais características do circuito, embora não apresente detalhes julgados irrelevantes e que só tornariam o simulador ainda mais lento e complicado.

Devendo trabalhar com interrupção, o simulador deverá conter um parâmetro que represente, no computador hospedeiro,

o tempo real do computador simulado. Este parâmetro foi implementado simplesmente como um contador de número de instruções executadas, representado portanto o número de "tempos médios de execução" utilizados, embora seja relativamente simples substituí-lo por um contador de tempo que seja atualizado de acordo com o tempo realmente utilizado na execução de cada instrução pelo Patinho Feio. Deve-se levar em conta, no entanto, que isto tornaria a "execução" mais lenta, recomendando-se para os casos em que se tenha realmente interesse no estudo destes parâmetros.

3.3.1 O Modelo do Sistema de Entrada e Saída.

A simulação da entrada e saída utiliza o seguinte modelo simplificado:

- uma palavra do HP 2116-B representa o"flipflop"de interrupção do sistema. É feita = 1 por um pedido de interrupção aceito e = Ø pela instrução PUL.

- os "flipflops" das interfaces são representados pelos bits de uma "palavra de estado" da interface a qual é por sua vez representada por uma palavra do HP 2116-B. Como existem , nas interfaces utilizadas, um número relativamente pequeno de"flipflops", o próprio registrador de dados ("buffer" da interface) também foi incluído nesta palavra. Assim, a primeira metade da palavra de estado contém informações relativas ao conteúdo dos "flipflops" da interface, e a outra metade representa o registrador de dados da mesma. Para que haja uniformidade de tratamento, bits em posições correspondentes de duas palavras de estado representam "flipflops" correspondentes das interfaces representadas. Caso uma das interfaces não utilize tal "flipflop", este bit não é tratado pelo programa de simulação, embora esteja fisicamente presente no modelo.

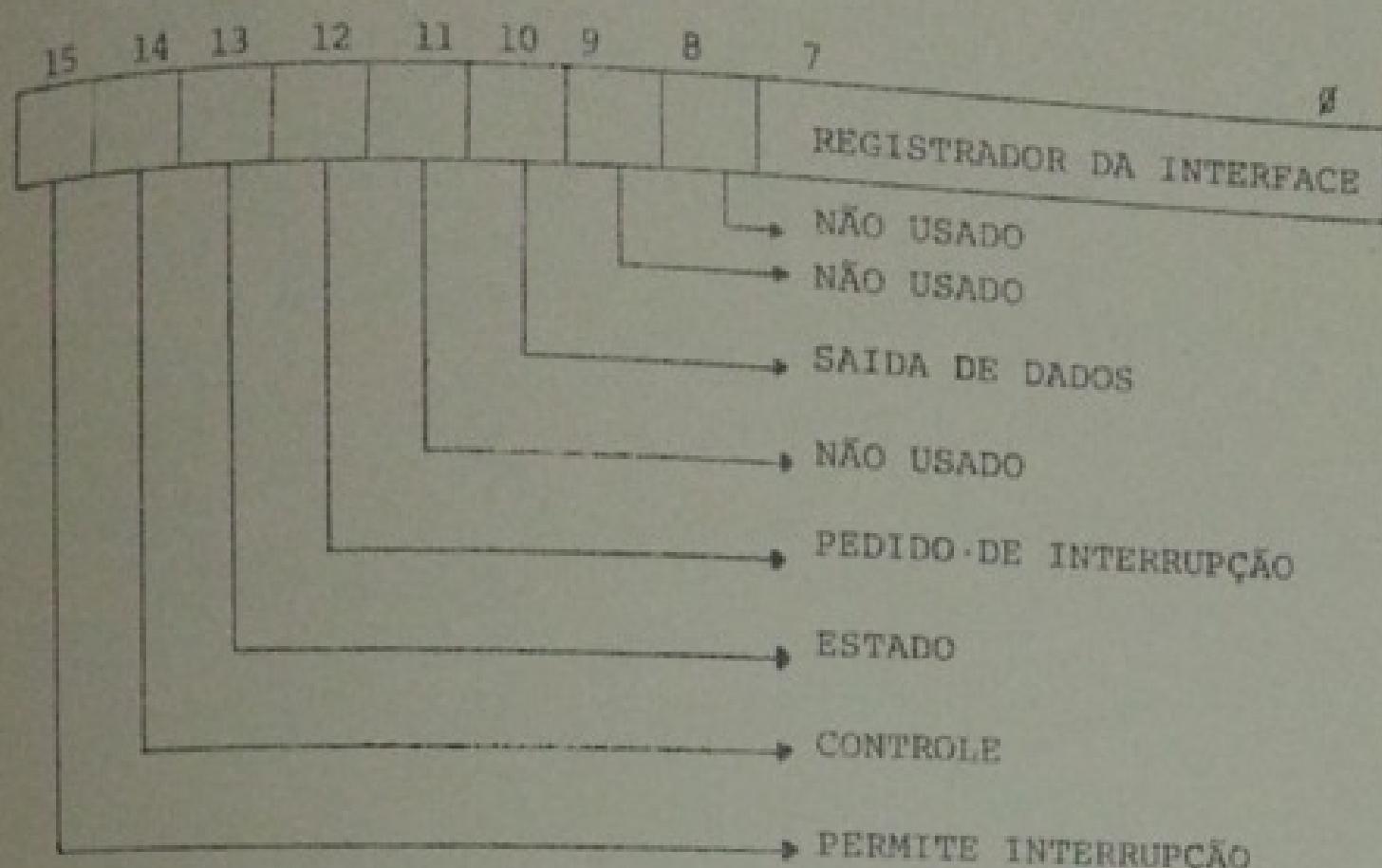



Fig. 3.3.1.1 - Representação de um Interface no Simulador. Os bits Ø a 7 representam o Registrador da Interface.

Os demais representam o estado dos diversos "flipflops" da mesma.

- o circuito de interrupção simplificado utilizadona simulação foi o da figura 3.3.1.2. Cada bloco "INTERFACE" foi simulado de acordo com o circuito da figura 3.3.1.3.

Fig. 3.3.1.2 - Modelo do sistema de Interrupção.

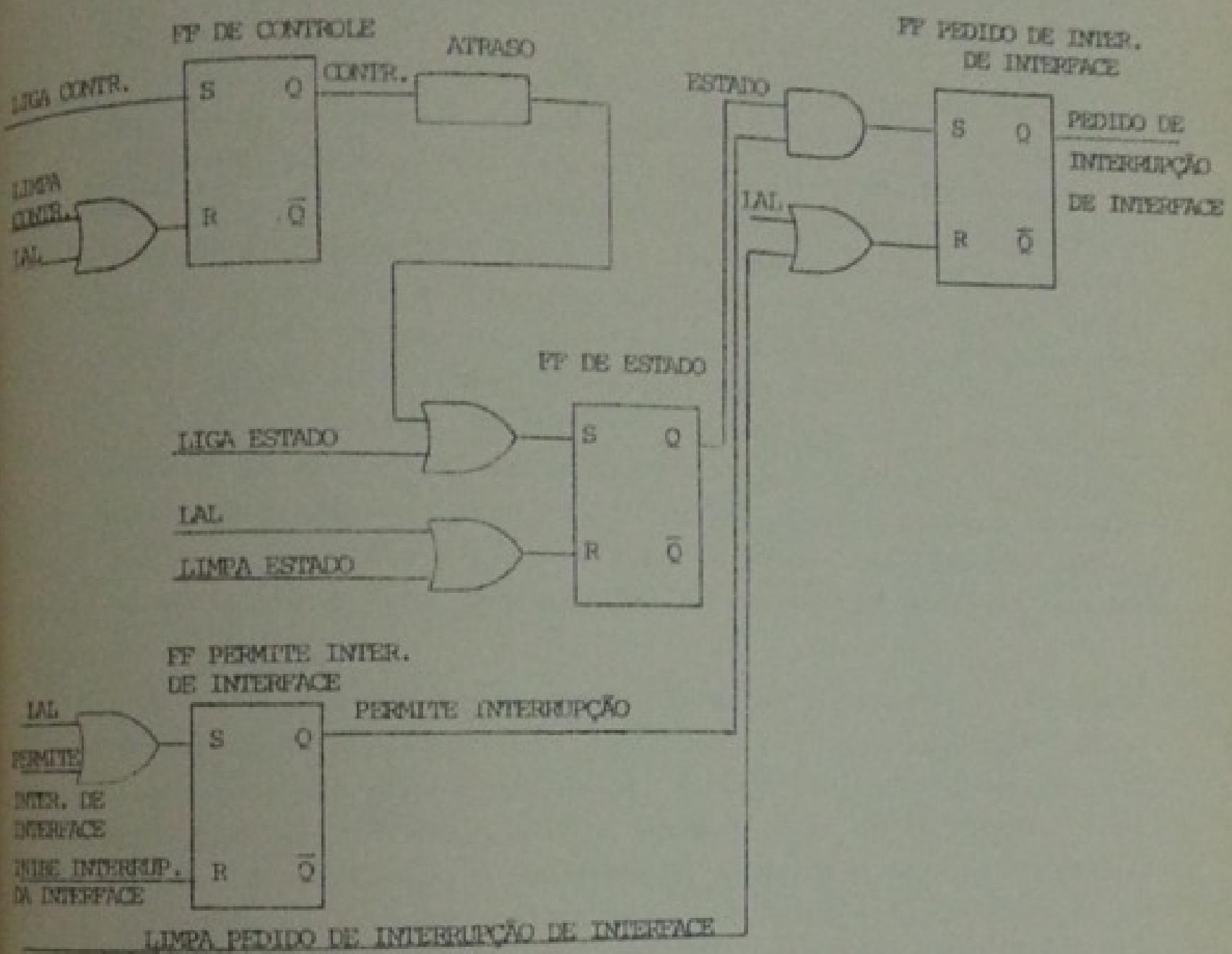



Fig. 3.3.1.3 - Modelo do circuito de interrupção da interface

- o sinal LAL foi simulado por um comando de console equivalente à ação de apertar o botão de "preparação".

3.3.2 A interação entre o interpretador de instruções e o simulador de entrada e saída.

Conforme foi descrito anteriormente, as instruções que
não manipulam entrada e saída são interpretadas de maneira rela
tivamente simples, pela execução de rotinas que implementam, no
modelo em nível de registradores, a função especificada. Com a
introdução de instruções de entrada e saída nessa simulação,tor
nou-se necessário levar em conta o tempo de execução das opera
ções de entrada e saída, bem como os detalhes, agora quase ao
nível de portas lógicas, dos circuitos de interrupção do siste
ma e das interfaces. Procurou-se modularizar as rotinas de tra
tamento de entrada e saída, bem como as de análise de interrup
ção permitindo um acesso relativamente simples a essas rotinas,
com o que se torna fácil acrescentar novas "interfaces", bem co
mo modificar o funcionamento das já existentes, quando necessá
rio.

A introdução da simulação da entrada e saída acarreta
uma série de consequências, tanto na fase de programação como
na execução do simulador, devido à característica dinâmica das
operações de entrada e saída. Assim, apesar de as rotinas bási
cas de interpretação e de execução da maioria das instruções per
manecerem inalteradas pela inclusão de simulação de entrada e
saída, o programa principal, isto é, o programa coordenador das
diversas operações de simulação perde a simplicidade que exibia
antes de tais rotinas serem implantadas, pois neste caso é ne
cessário incluir, por exemplo,parâmetros de tempo para contagem
de atrasos decorridos desde o acionamento de um dispositivo até
que a operação requisitada se complete (um para cada dispositi
vo), além de tabelas de bits de estado que representam os
"flipflops" de cada interface e seus registradores, bits de es
tado do sistema que representam os "flipflops" de "interrompi
do", "espera" e "pedido de interrupção".

Como todos estes parâmetros devem ser testados e atua
lizados antes da execução de cada instrução, uma das consequên
cias mais sérias de sua inclusão é a de tornar mais lento o pro
cesso de simulação. Para atenuar o problema, introduziu-se um
comando de console que permite suprimir os testes de interrup

ção no caso da execução de programas que não utilizam o sistema de interrupção. Além disso, foi incluído o teste de uma chave de painel, segundo a qual é possível limitar o número de ciclos necessários para que o simulador de entrada e saída responda ao acionamento de um dispositivo. Trata-se de tornar a entrada ou saída simulada mais rápida em relação ao processador central que a entrada ou saída real, se isto for solicitado. Deste modo, em programas onde não houver necessidade de manter as relações entre o tempo de processamento e o tempo de entrada e saída, é possível tornar o processamento de simulação, da entrada/saída até 2500 vezes mais rápido, em alguns casos. Isto é obtido, no simulador, simplesmente tornando o periférico simulado muito mais rápido em relação ao processador central que o periférico real, pela modificação do parâmetro de tempo relativo correspondente ao periférico em questão. Quando houver necessidade, bastará solicitar que se cumpram as verdadeiras relações entre os diversos tempos para que as velocidades relativas entre os periféricos e o processador central, bem como as relações entre as velocidades dos periféricos sejam mantidas, como é normalmente necessário durante a fase de depuração de programas com interrupção que utilizam simultaneamente vários periféricos com sobreposição ("overlap") de processamento e de entradas e/ou saídas em periféricos de velocidades diferentes.

3.4 O programa Controlador

O programa principal, juntamente com a rotina de atendimento de comandos de console, formam o programa controlador das atividades do simulador-interpretador.

Sua função principal é naturalmente a de coordenar a sequencialização das chamadas das diversas rotinas de simulação. Na fig. 3.4.1 vê-se um fluxograma resumido do simulador-interpretador, no qual pode ser notado facilmente estar o programa dividido em duas partes quase independentes: a fase de console e a fase de execução.

3.4.1 A fase de Console

Esta fase corresponde ao estado em que a máquina não está executando programas, isto é, ao estado em que é permitido interferir nos registradores, memória, etc, pelos controles do pai

Fig. 3.4.1 - Diagrama de Blocos Resumo do Simulador-Interpretador

nel. Além disto, incluiu-se alguns recursos adicionais nesta fase para que se torne mais simples a depuração de programas. A fase de Console é acionada todas as vezes em que isto for solicitado pelo painel, bem como no início da execução do programa e em caso de parada de processamento decorrente da simulação da instrução PARE pela fase de execução. Por outro lado, a fase de console passa o comando à fase de execução apenas quando um comando de Partida fôr fornecido. Na fase de Console são interpretados e executados os seguintes comandos:

A - Armazenamento: Sintaxe: $A,(XX,)_1^n$

Este comando permite o armazenamento de n constantes hexadecimais XX de dois caracteres em posições consecutivas a partir de CI (contador de instruções) atual obtido pela sequência normal do processamento ou por um comando E. A execução deste comando altera o valor do CI para CI + n.

Exemplo: A, 21, 1Ø, F3, 5A,

C - Configuração: Sintaxe: C

Este comando faz com que o simulador aceite, pela console, a reconfiguração dos periféricos, permitindo que se forneça o número dos canais do Patinho Feio, correspondentes à impressora, à perfuradora de fita e à TTY do HP-2116-B.

D - "Dump": Sintaxe: D, XXX, YYY

Este comando faz com que seja gerado um "dump" de memória na impressora. Este "dump" exibe o conteúdo de todas as posições de memória no intervalo XXX a YYY. XXX deve ser menor que YYY e ambos são constantes hexadecimais.

Exemplo: D, 1ØØ, 3FF

E - <u>Endereçamento</u>: Sintaxe: E, XXX

Este comando preenche a variável CI (contador de instruções) com uma constante XXX dada em hexadecimal, fazendo com que seja apontada a próxima instrução a executar, a expor ou a preencher com o comando de armazenamento.

Exemplo: E, F8Ø

I - <u>Inibe tratamento de interrupção</u>: Sintaxe: I

Este comando faz com que a rotina de testes de interrupção não seja executada. Serve para acelerar a execução em caso de o programa a executar não utilizar o sistema de interrupção.

L - <u>Limpa</u>: Sintaxe: L

Equivale ao botão de "preparação" do computador.

- Limpa todos os bits de controle das interfaces, bem como os pedidos de interrupção e o "flipflop" de interrupção do sistema.
- Liga todos os "flipflops" de estado das interfaces.
- Liga o "flipflop" de Permite Interrupção do sistema, e inibe os das interfaces.
- Anula a ação de um comando I anterior.

M - <u>Mensagem</u>: Sintaxe: M

Liga a opção de impressão da mensagem "PATO EM ESPERA" na console quando fôr executada uma instrução ESP.

N - <u>Suprime Mensagem</u>: Sintaxe: N

Desliga a opção de impressão de mensagem "PATO EM ESPERA"

P - Partida: Sintaxe: P

Transfere o controle para a fase de execução do simulador. A primeira instrução a executar é a apontada pelo CI (contador de instruções).

S - Carrega "Pré-Loader": Sintaxe: S

Carrega, a partir da posição 4 da memória simulada, o "pré--loader", programa com o qual é possível carregar o "bootstrap", programa residente que carrega todos os programas de uma fita objeto absoluta para a memória.

T - "Trace": Sintaxe: T (, XXX, YYY)$^{1}_{0}$

Liga a opção de "TRACE" no intervalo XXX, YYY se este for especificado. É permitida a declaração de até 10 comandos T com intervalos a escolher. Se for fornecido o comando T sem operandos, será ligada a opção "trace" total.

A execução deste comando está vinculada ao estado da chave 15 do painel. Enquanto esta estiver ligada, será fornecido o "trace" nas regiões desejadas. No caso oposto, o "trace" não será fornecido.

Exemplo: T, 1ØØ, 5ØØ

X - Exposição: Sintaxe: X, YY

Este comando solicita ao simulador a impressão, na console, de YY posições de memória a partir da posição dada pelo CI. YY é um número hexadecimal.

A execução deste comando altera o valor do CI para CI + YY:

Exemplo: X, 1Ø

3.4.2. A fase de execução

Recebido um comando de Partida da fase de console,passa-se para a fase de execução. Antes de mais nada ocorre o teste dos parâmetros de interrupção, bem como a atualização dos parâmetros de tempo de entrada e saída (estes testes são suprimidos pelo comando I da fase de console). A seguir, é executada a busca da proxima instrução apontada pelo CI. Se for uma instrução de entrada e saída, a ação a executar é a de simplesmente acertar alguns parâmetros, correspondentes aos "flipflops" da interface correspondente. Caso seja uma instrução PARE, a rotina da fase de CONSOLE será ativada. Se for qualquer outra instrução válida,inicia-se a sua simulação.Após a execução de qualquer instrução, é testado se a mesma está ou não no intervalo de "trace" se a chave 15 estiver ligada. Caso contrário, a próxima instrução será simulada segundo a mesma sequência.

Após a execução de qualquer instrução, é testada também a chave 14, que, se ligada, indica que se deseja retornar o controle à fase de Console.

A execução propriamente dita é trivial,constando apenas de um reconhecimento da instrução a ser executada, seguida de um desvio para a rotina de execução correspondente. Esta consta de uma sequência de instruções do HP-2116-B que, como foi dito,executa, nas variáveis do modelo utilizado, a mesma ação global que a instrução simulada executaria nos registradores e posições de memória correspondentes do Patinho Feio.

Apenas no caso das instruções de referência à memória, o processo é diferente pois há necessidade de cálculo do endereço efetivo do operando, o qual pode ser efetuado de diversas maneiras, como endereçamento direto, indexado,indireto e indireto indexado. Na fase de execução deteta-se a tentativa de execução de instruções ilegais, caso em que é fornecida uma mensagem de erro

3.5. Comentários, críticas e sugestões

A análise dos resultados obtidos nas inúmeras utilizações do programa simulador-interpretador implementado, bem como a avaliação de seu desempenho por meio do estudo da sua implementação permitiu que se chegasse às seguintes conclusões:

- É muito útil durante a fase de depuração de programas, especialmente quando as dimensões são tais que tornam impossível a utilização do programa de depuração (cap. 5) no próprio Patinho Feio.

- A disponibilidade de uma impressora de linhas no sistema hopedeiro (HP-2116-B) torna o programa simulador bastante indicado para substituir o Patinho Feio na execução de programas onde o volume de saída impressa seja grande, sempre que na configuração do mesmo não estiver incluída uma impressora.

- É a única maneira prática disponível para o acompanhamento de programas que utilizam o sistema de interrupção do computador. Isto o torna insubstituível no caso da depuração e acompanhamento de rotinas tratadoras de interrupção, programas controladores de entrada e saída, etc. (ref. 2). Para facilitar ainda mais o acompanhamento dos eventos relacionados com a entrada e saída, criou-se uma saída opcional que imprime o estado dos "flipflops" de interrupção do computador e das interfaces (chave 11).

- É bastante útil quando a natureza do programa que se está depurando for tal que exija uma frequente utilização de "dumps", "traces", etc. Estas opções, quando utilizadas no Patinho Feio, exigem que parte da memória seja ocupada pelas rotinas correspondentes, o que impede que programas de grande extensão as utilizem. Para casos como êste, o uso do simulador é indicado, uma vez que tais rotinas não ficam na memória simulada do Patinho Feio, mas fazem parte do próprio programa de controle do simulador-interpretador.

- Em matéria de velocidade, o interpretador é, pela sua própria natureza, bastante lento em relação ao Patinho Feio. Esta limitação foi bastante atenuada evitando-se totalmente o uso da memória de massa do HP-2116-B no simulador. O programa foi escrito como um módulo único (não segmentado), e a memória principal do Patinho Feio foi simulada totalmente na memória principal do HP-2116-B. Para efetuar a simulação de mais memória, será necessário que se inclua, no programa, uma série de rotinas da paginação

que permitirião a utilização da memória de massa do HP-2116-B para simular a memória principal do Patinho Feio. Êste processo deverá ser utilizado quando for necessário desenvolver programas com mais de 4096 palavras para uma versão do Patinho Feio com memória expandida. A eficiência do algoritmo de paginação dependerá, no caso, da política de substituição das páginas residentes na memória, que influirá decisivamente na velocidade da simulação. Para que seja projetado um esquema eficiente, será necessário desenvolver previamente um estudo estatístico do comportamento dos programas já implementados, bem como dos que serão desenvolvidos especialmente com esta finalidade. Com base nestes estudos, será possível projetar um programa de simulação adequado aos programas implementados para o Patinho Feio. Êste assunto, bastante complexo e ainda não totalmente elucidado, será estudado oportunamente pela equipe de desenvolvimento do Laboratório de Sistemas Digitais, não cabendo aqui maiores considerações a respeito.

- Como a maioria dos processos de entrada e saída são afetados por fatores de caráter aleatório, e como tais fatores não foram considerados no modelo utilizado, o simulador não é suficiente, em alguns casos, para detetar todos os problemas de um dado programa. É o que ocorre com programas que, apesar de exibir funcionamento satisfatório no simulador, apresentam problemas intermitentes quando executados no Patinho Feio. É, pois, indipensável o teste final dos programas de interrupção no próprio Patinho Feio, para que se possa garantir seu funcionamento apesar do caráter aleatório dos processos de entrada e saída.

- Em programas simuladores como o aqui descrito, é normalmente conveniente a inclusão de rotinas de contabilização, para efeito de levantamento estatístico da utilização dos diversos recursos do simulador, inclusive de cada instrução em particular. Estas rotinas poderão atuar tanto no âmbito do programa particular que está sendo simulado, como no âmbito global, em que se faz a contabilização completa de todos os programas já processados. É desejável que alguns parâmetros sejam contabilizados, tais como:

a) frequência de execução de cada instrução em particular;
b) frequência de execução de cada grupo de instruções;
c) frequência de execução de sequências de instruções pré--determinadas;

d) tempo de processamento de cada programa;
e) tempo de entrada e saída de cada programa;
f) relação entre o tempo de processamento e o de entrada e saída;
g) relação entre o tempo total de processamento e o tempo de atendimento de interrupções em cada programa;
h) número de instruções executadas em cada programa;
i) número de solicitações de entrada e saída em cada dispositivo;
j) tempo de resposta das interrupções ocorridas.

A versão atual do simulador não inclui qualquer dessas opções, que poderão vir a ser incorporadas quando da adaptação do mesmo à ampliação de memória. Com a inclusão de algumas destas rotinas, será possível o estudo de um conjunto de instruções mais eficiente, por meio da implementação de conjuntos das instruções experimentais, que poderão auxiliar no projeto de novas versões do Patinho Feio ou mesmo de outras máquinas. Para tanto, é necessário que se incluam alguns recursos adicionais que facilitem a mecanização do processo de programação das rotinas de simulação. Uma sugestão é a de implementar-se um compilador para uma linguagem de descrição de um conjunto de instruções, na qual possam ser especificados, em detalhes, os formatos das instruções e o modo pelo qual estas instruções são executadas. A saída de tal compilador poderá ser, por exemplo, um programa que simula o computador descrito.

- Outra característica que poderia ser melhorada numa futura versão do simulador é o conjunto de comandos de console. Sugere-se a seguir alguns novos comandos desejáveis:

a) Carregar Programa - Êste comando teria como finalidade carregar, na memória simulada do Patinho Feio, um programa - objeto em formato absoluto como o gerado pelo montador absoluto. A vantagem dêste comando é a de dispensar o operador do trabalho de carregar o "bootstrap" (carregador absoluto), além de tornar a operação de carga de programas muito mais rápida.

b) Geração de Fita Binária Carregável - Esta operação seria equivalente à do "dump" binário, atualmente executado por

um programa gerador de fita carregável. Para êste comando, é interessante que se forneça dois parâmetros, os endereços incial e final da memória que se deseja descarregar para fita perfurada. A vantagem dêste comando é a de não utilizar memória simulada do Patinho Feio, além de tornar simples e rápido o processo de geração da fita binária carregável.

c) Salvar o Estado do Simulador - Tendo sido o simulador implementado em um computador cujo sistema operacional permite a execução de um único programa por vez, é muito conveniente que um processo de simulação, que é às vêzes muito demorado, possa ser interrompido para dar lugar a um outro programa mais prioritário. Para isto ser possível, é preciso guardar as informações do estado em que o simulador se encontrava quando ocorreu a interrupção, afim de que, quando for possível, o programa possa ser reiniciado a partir do ponto em que a interrupção ocorreu sem a necessidade de reexecutar toda a simulação já efetuada. Basta para tanto que sejam guardados, num arquivo em disco ou fita magnética, todas as variáveis do programa simulador que representam estados de máquina simulada. No caso particular de implementação aqui descrita, é suficiente salvar a área de "common", onde estão guardados todos os registradores, memória e estados das interfaces simuladas, desde que se assegure que não haja nenhuma entrada ou saída em andamento.

d) Restaurar o Estado do Simulador - Quando se usar o comando de Salvar o Estado do Simulador, um arquivo será preenchido com informações sobre o estado da simulação no instante em que o mesmo foi interrompido. A seguir, outros programas quaisquer poderão ser executados, até que se deseje reiniciar a simulação. Para isto, deve-se iniciar a execução do simulador, e logo a seguir, executado o comando de Restaurar o Estado do Simulador. Êste comando deverá buscar no arquivo gerado pelo comando de Salvar Estado do Simulador as informações necessárias para a atualização das variáveis do simulador, bastando que seja fornecido um comando de Partida para que a simulação continue a partir do ponto onde tiver sido interrompida.

e) Proteção de Memória - Às vêzes é conveniente que se saiba se uma certa região de memória é modificada, invadida ou mesmo referenciada por um programa durante sua execução. É relativa

mente simples efetuar-se um teste para verificar a ocorrência de tais eventos, bastando para isto que, antes da execução de qualquer instrução, seja consultada uma tabela de limites protegidos de memória (a exemplo do que se faz na rotina de teste de limites de "trace"). Deverá também ser testado, nesta ocasião, o tipo de violação ocorrido (tentativa de modificação, invasão ou referência), sendo o controle neste ponto devolvido à console para decisão, por parte do operador, da ação a ser tomada. Uma consequência da inclusão deste teste é a redução da velocidade do processamento.

A sintaxe dêste comando deverá incluir três parâmetros:

1) tipo de proteção - modificação, invasão ou referência.
2) limite inferior - primeiro endereço coberto pela proteção.
3) limite superior - último endereço coberto pela proteção.

f) Deteção da Execução de uma dada Instrução - Durante a fase de depuração de programas, é comum que algumas posições de memória sejam alteradas, pela inclusão forçada de comandos de parada, com a finalidade de obrigar o computador a devolver o controle ao operador quando o programa passar pela instrução alterada. O trabalho do operador ficará bastante reduzido se for incluída uma opção que associe a cada instrução uma variável que, se ligada, faz com que o controle passe à console assim que a instrução for decodificada (ou executada, dependendo da conveniência). Com esta opção será também possível proteger o programa contra a execução de determinados grupos de instruções, evitando assim que sejam danificados processos de entrada e saída, conteúdo de registradores e assim por diante. A inclusão dêste teste resultará também numa perda em velocidade do simulador.

A sintaxe mais recomendável para êste comando, levando em conta a facilidade de implementação e a convivência do operador é aquela em que se inclui como parâmetro o código de máquina da instrução que se deseja detetar.

CAPÍTULO 4. UM DESMONTADOR PARA A LINGUAGEM DE MÁQUINA DO PATINHO FEIO

4. UM DESMONTADOR PARA A LINGUAGEM DE MÁQUINA DO PATINHO FEIO

Quando se tem em mãos um computador de pequeno porte como o Patinho Feio, com todas as suas dificuldades de utilização, é muito comum que o trabalho investido para se obter uma nova versão de um programa em teste seja relativamente grande, obtendo-se como resultado uma nova fita objeto ligeiramente diferente da última versão em teste, uma vez que normalmente as correções efetuadas são de pequena amplitude. Isto ocorre frequentemente na fase de depuração do programa, quando, dependendo dos erros detetados, é mais fácil a correção do programa pela interferência direta do programador no conteúdo da memória que uma nova montagem do mesmo, à vêzes executada em outro computador. Com a finalidade de evitar tal trabalho adicional, o programador pode corrigir o programa diretamente na memória do computador, e, em seguida, continuar a testar seu programa. Uma vantagem de tal operação é, às vêzes, uma deteção mais rápida de erros no programa, o qual poderá ficar pronto mais rapidamente. Naturalmente, as correções devem ser anotadas para uma futura modificação do programa fonte. Além disso, é recomendável que, cada vez que modificações deste tipo foram feitas, uma fita objeto ("dump" de memória) seja gerada para possibilitar uma posterior recarga do programa corrigido. Em situações como a que acaba de ser descrita, muitas vêzes são geradas fitas objeto parciais, isto é, que não englobam todo o programa, ou, ainda, versões que funcionam, de um programa ainda em desenvolvimento, mas que não possuem um programa fonte correspondente. É este também um caso típico de programas objeto gerados por compiladores de uma linguagem de alto nível.

Em qualquer dos casos, pode tornar-se importante conhecer o conteúdo de uma fita objeto, quer para se reconstituir uma documentação perdida, quer para se ter maior facilidade de deteção de erros de lógica em um compilador em desenvolvimento. Seja qual for a finalidade a que se destine, um programa que automatize a tarefa de interpretar o conteúdo de uma fita objeto é de grande utilidade em tais casos. Um programa que executa tal tarefa, via de regra conhecido impropriamente como "descompilador", é chamado desmontador ("disassembler"), e é o objeto deste capítulo.

A existência de dois formatos de fita objeto, o absoluto e o relocável, impõe a necessidade das duas versões correspondentes para o desmontador. O desmontador absoluto é útil quando se

tem em mãos uma fita objeto absoluta de conteúdo desconhecido, ou então um "dump" de memória em formato objeto absoluto carregável. O desmontador relocável tem utilidade análoga nos programas objeto relocáveis e em saídas de compiladores.

4.1. Especificações do desmontador

Quando se pensa em construir um desmontador, é necessário levar-se em conta até que ponto se deseja reconstituir o programa fonte a partir do programa objeto binário. Deve-se considerar, por exemplo, que o máximo que um programa dêste tipo pode fornecer é um programa fonte tal que, fornecido como dado ao montador, faça com que este reproduza a fita objeto a partir da qual este programa fonte foi obtido. É de se esperar que um mesmo programa objeto possa ser gerado a partir de inúmeros programas fonte diferentes. Basta que estes programas sejam tais que as suas instruções gerem a mesma sequência de palavras binárias. Um exemplo trivial de dois programas que geram o mesmo código é aquele em que os programas têm a mesma sequência de instruçõesecujos rótulos sejam correspondentes mas diferentes entre si (fig.4.1.1).

ORG /1Ø	ORG /1Ø
CAR X	CAR Y
ARM A	ARM B
PARE	PARE
X DEFC Ø	Y DEFC Ø
A DEFC Ø	B DEFC Ø
FIM /1Ø	FIM /1Ø

Fig. 4.1.1 - Exemplo trivial de dois programas que geram o mesmo código objeto

Um elemento do programa fonte, que normalmente fica perdido após a montagem, é o conjunto de símbolos utilizados como rótulos no programa. A não ser que o montador registre na fita perfurada o conteúdo da tabela de símbolos, esta certamente deixa de

existir uma vez que o código objeto esteja totalmente gerado. Como, no presente caso, não é executado tal procedimento, os rótulos não são disponíveis para a desmontagem, devendo ser gerados pelo programa em caso de necessidade. Alguns rótulos, entretanto, são mantidos, após a montagem, na própria fita perfurada. São os símbolos globais, declarados como externos pela pseudo EXT.Estes símbolos devem ser preservados, e, na desmontagem, deverão aparecer tais como definidos no programa fonte que deu origem à fita objeto em questão.

Um programa absoluto qualquer pode ser obtido montando-se uma sequência de pseudos DEFC, isto é, constantes cujos valores devem ser iguais aos códigos de máquina das instruções a elas correspondentes (fig. 4.1.2).

end.		
	ORG /1Ø	ORG /1Ø
10	CAR X	DEFC /4Ø DEFC /15
12	ARM A	DEFC /2Ø DEFC /16
14	PARE	DEFC /9D
15	X DEFC Ø	DEFC Ø
16	A DEFC Ø	DEFC Ø
	FIM /1Ø	FIM /Ø1Ø

Fig. 4.1.2 - Geração de programa fonte absoluto composto apenas de constantes

Naturalmente uma desmontagem deste tipo não é útil quando se deseja conhecer o conteúdo lógico do programa, correspondendo apenas a uma espécie de "dump" de memória. Deseja-se portanto do programa desmontador, que ao menos sejam listados os mnemônicos correspondentes às diversas instruções. É aqui que surge o primeiro problema dos desmontadores: a distinção entre instruções absolutas e constantes. Este problema não existiria se o programa objeto contivesse a informação adicional em questão, o

que é impossível no caso de um "dump" de memória. Consequentemen te, será do usuário a tarefa de descobrir, a partir da análise da lógica do programa, se uma palavra é uma constante ou uma instrução.

No caso de computadores com instruções de comprimento não uniforme, como o Patinho Feio, aparece um outro proble ma intimamente relacionado com este. Trata-se de saber exatamen te onde começam os trechos de programa e os de dados.

ORG /1Ø		ORG /1Ø		ORG /1Ø		ORG /1Ø	
PLA /8Ø	ØØ8Ø	DEFC Ø	ØØ	DEFC Ø	ØØ	PLA /8Ø	ØØ8Ø
ARM /1Ø	2Ø1Ø	DEFC /8Ø	8Ø	LIMPØ	8Ø	DEFC /2Ø	2Ø
PARE	9D	DEFC /2Ø	2Ø	ARM /1Ø	2Ø 1Ø	PLAX /9D	1Ø9D
FIM		PLAX /9D	1Ø9D	PARE	9D	FIM	
		FIM		FIM			
(a)		(b)		(c)		(d)	

Fig. 4.1.3 - Exemplo de trechos diferentes de programa com o mesmo código objeto, com problema de ambiguidade na desmontagem.

Na fig. 4.1.3 observa-se um caso como o descrito. Como se pode observar facilmente, um mesmo programa absoluto não mui to extenso pode ser interpretado de tantas maneiras diferentes que é totalmente inviável tentar-se encontrar em algoritmo que automatize a geração da sequência correta de mnemônicos a ele correspondentes. Como esta geração só pode ser feita, num caso como este, mediante decisões humanas, baseadas na lógica do pro grama, foi imposta uma restrição drástica quanto à classe de pro gramas em que o desmontador produz código fonte correto: garan te-se somente que o desmontador interpreta corretamente trechos de código correspondentes a áreas de programa, contendo apenas instruções de máquina em sequência. Nada se pode garantir quan do houver alguma declaração de constante entre as instruções.

Chega-se desta maneira a uma especificação mo desta para o desmontador absoluto; sua ação deve consistir apenas em decodificar a linguagem de má quina para linguagem mnemônica, admitindo ser o

trecho a desmontar constituído de instruções apenas, não contendo dados.

Segundo esta especificação, foi implementada a primeira versão do desmontador absoluto. Sua saída imprensa consistia de uma coluna de mnemônicos associados ao código objeto, e de uma coluna com os operandos correspondentes em hexadecimal.

Sendo mesmo assim difícil a utilização de saída do desmontador, por causa da ilegibilidade associada a uma listagem dêste tipo, decidiu-se criar uma segunda versão do desmontador absoluto, onde as referências à memória são representadas simbolicamente, ficando a listagem com três colunas: a dos rótulos , onde são definidos os símbolos referenciados nos campos de operando das instruções de referência à memória encontradas no programa, a dos mnemônicos, idêntica à da primeira versão, e a dos operandos, que consta agora de símbolos ou números em hexadecimal, conforme seja a instrução de referência à memória ou não, respectivamente.

Para o uso do desmontador relocável, as especificações podem ser um pouco menos tolerantes, uma vez que as informações adicionais de relocação contidas na fita objeto relocável permitem que se chegue muito mais próximo do programa fonte que na versão absoluta. Em outras palavras, não há ambiguidade em um texto objeto relocável, sendo possível reconstituir com muito maior fidelidade o programa fonte nêste caso. Por isso, exige-se do desmontador relocável a produção de um programa fonte que seja idêntico ao que gerou a fita objeto correspondente, a menos dos rótulos e das pseudo instruções sinônimas de instruções de referência à memória (DEFE, DEFI), as quais são interpretadas como instrução de referência à memória (PLA e PLAX, respectivamente).

O problema da pseudo DEFC, que era crucial no caso do desmontador absoluto, torna-se totalmente irrelevante neste caso, pois o código objeto relocável correspondente a uma instrução qualquer difere do código correspondente gerado por uma pseudo DEFC equivalente, o que elimina o problema.

4.2 A lógica do desmontador

Uma vez estabelecidas as restrições aceitáveis para o desmontador absoluto, e constatado que um programa desmontador que apresente tais restrições ainda é útil para o programador, passou-se a projetar a lógica do mesmo.

Como se sabe, os programas objeto absolutos apresentam-se no formato de sequência de blocos de dados (apêndice 3 ; cap. 2); é a partir das informações contidas nestes blocos que o desmontador constrói o programa fonte, segundo a lógica descrita a seguir.

Primeiramente são dados valores iniciais às variáveis do programa, sendo, a seguir, executada a operação de desmontagem propriamente dita. Como foi discutido anteriormente, deseja-se do desmontador tanto a geração de mnemônicos associados ao código de máquina que está sendo tratado, como também a de rótulos simbólicos, tanto no campo de rótulos (no caso de a posição correspondente ter sido referenciada no programa), como no campo de operandos de instruções de referência à memória. Assim, é preciso que o programa objeto seja lido uma vez para que seja efetuada a montagem de uma tabela de endereços referenciados, e uma segunda vez para a geração do programa fonte a partir do programa objeto e da tabela anteriormente montada, sendo, pois, o trabalho de desmontagem executado em dois passos.

Descreve-se a seguir a lógica dos dois passos do desmontador relocável apenas, devendo-se ter em mente que no caso do desmontador absoluto valem as restrições descritas em 4.1, sendo todo o conteúdo da fita considerado como se fosse constituído apenas de instruções. Do ponto de vista da lógica do programa, o desmontador absoluto pode ser considerado como um caso particular do desmontador relocável, motivo pelo qual a sua descrição não será aqui apresentada. Deve-se observar, no entanto, que por facilidade de implementação, os dois desmontadores foram desenvolvidos separadamente, tendo em comum apenas as rotinas básicas de geração de rótulos, mnemônicos e operandos, sendo estas controladas por programas principais totalmente diferentes.

4.2.1 O primeiro passo do desmontador

A finalidade principal do primeiro passo do desmontador é a de gerar, a partir da fita objeto, uma tabela de todos os endereços referenciados pelas instruções do programa. Para isto, o programa deve ser varrido, instrução por instrução, identificando-se as instruções de referência à memória, de onde é extraído o campo dos operandos, o qual é colocado numa tabela de endereços se já não estiver nela contido. Feito isto para todas as instruções do programa, estará construída a tabela dos endereços referenciados pelo mesmo, a qual deverá ser utilizada pelo segundo passo para a geração dos rótulos onde fôr necessário. Nesta tabela, além do endereço propriamente dito, deverá constar também a informação de relocação, bem como um "bit" de definição, a ser utilizado pelo segundo passo (fig. 4.2.1.1)

D/I - Indica se o rótulo correspondente ao endereço já foi gerado.

R - Informações de relocação do endereço.

E .- Endereço propriamente dito.

Fig. 4.2.1.1 - Um elemento da tabela de endereços referenciados do desmontador

O programa de construção desta tabela é trivial. Para cada operando de instrução de referência à memória, é executada uma pesquisa na parte já construída da tabela, seguida de uma inclusão do novo endereço na mesma se este não for encontrado.

O bit D/I é feito "Indefinido", preparando a tabela para o segundo passo (fig. 4.2.1.2).

4.2.2 O segundo passo do desmontador

Uma vez construída a tabela dos endereços referenciados, no primeiro passo, o programa objeto é lido uma segunda vez, com o objetivo de fornecer informações para a geração dos mnemônicos e dos operandos. Os rótulos são ou não gerados, dependendo da presença ou não dos endereços a eles correspondentes na tabela de endereços referenciados. Em outras palavras, são gerados rótulos apenas para os mnemônicos correspondentes a instruções ou dados cujos endereços foram referenciados no programa.

O segundo passo do desmontador, cujo funcionamento será aqui descrito, tem como finalidade principal a geração, a partir de um código objeto fornecido, de um programa fonte, na linguagem do montador, tal que o código objeto a ele correspondente, produzido pelo montador, seja equivalente ao fornecido ao desmontador (fig. 4.2.2.1)

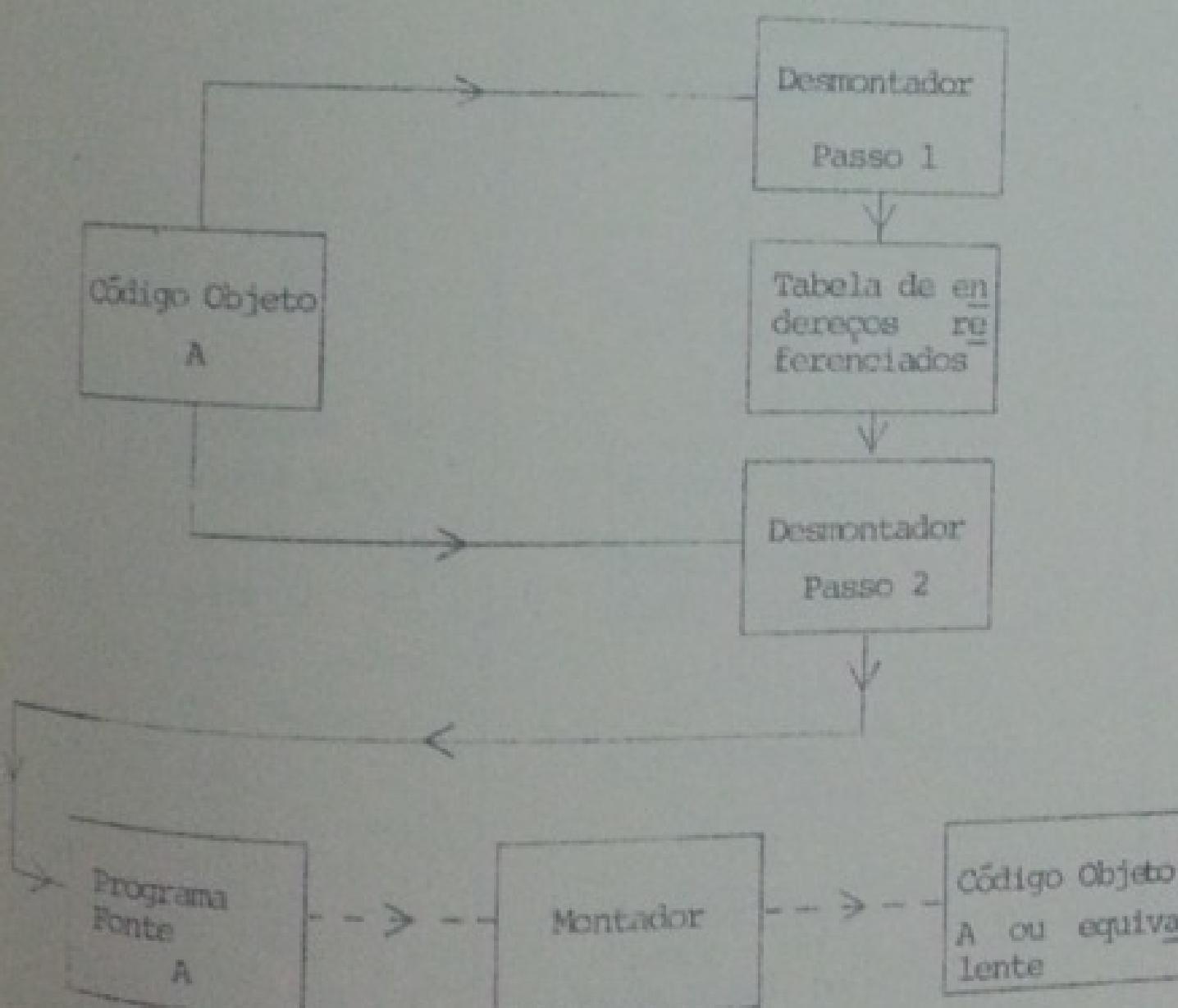



FIG. 4.2.2.1 - Funcionamento do Desmontador

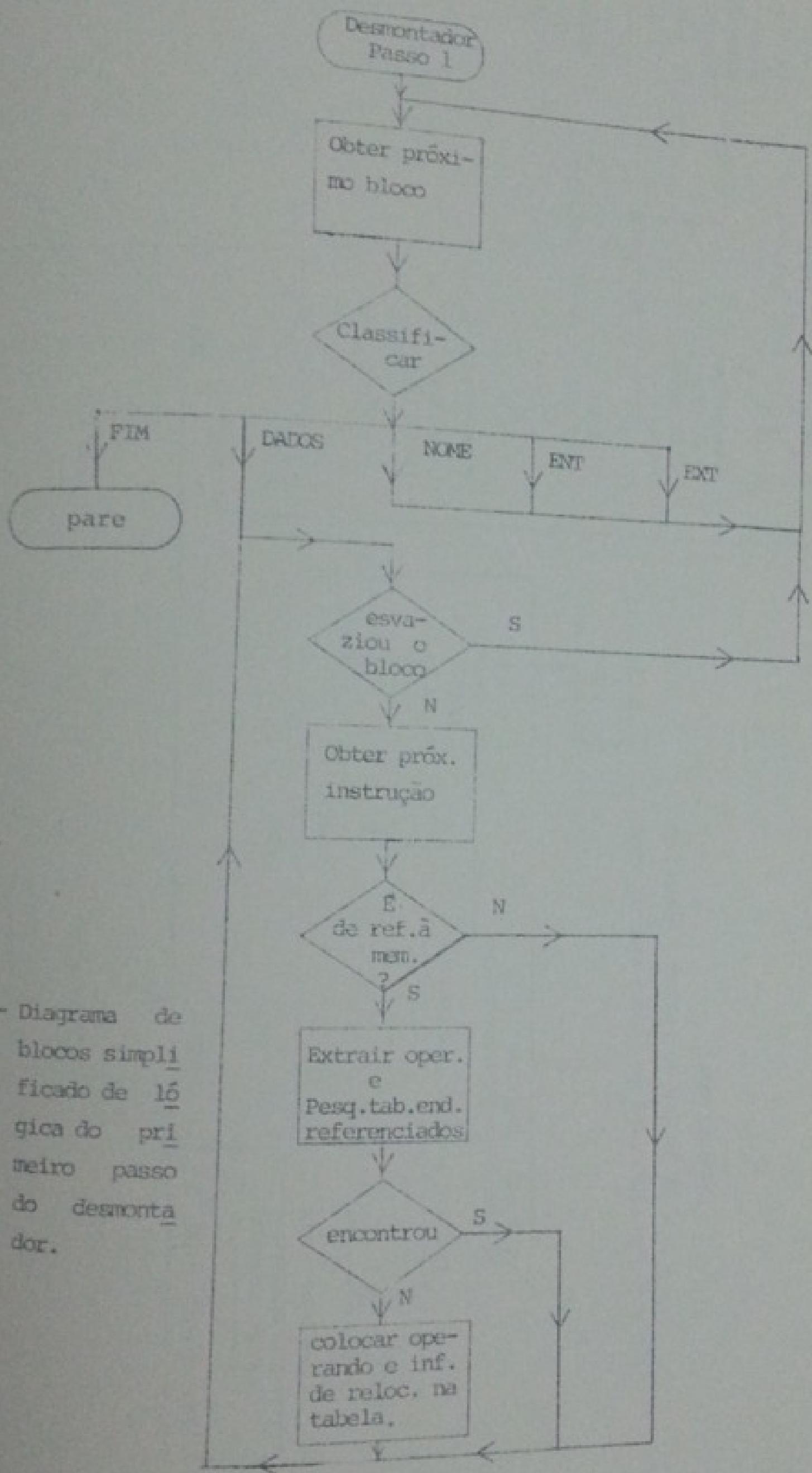



Fig. 4.2.1.2 - Diagrama de blocos simplificado de lógica do primeiro passo do desmontador.

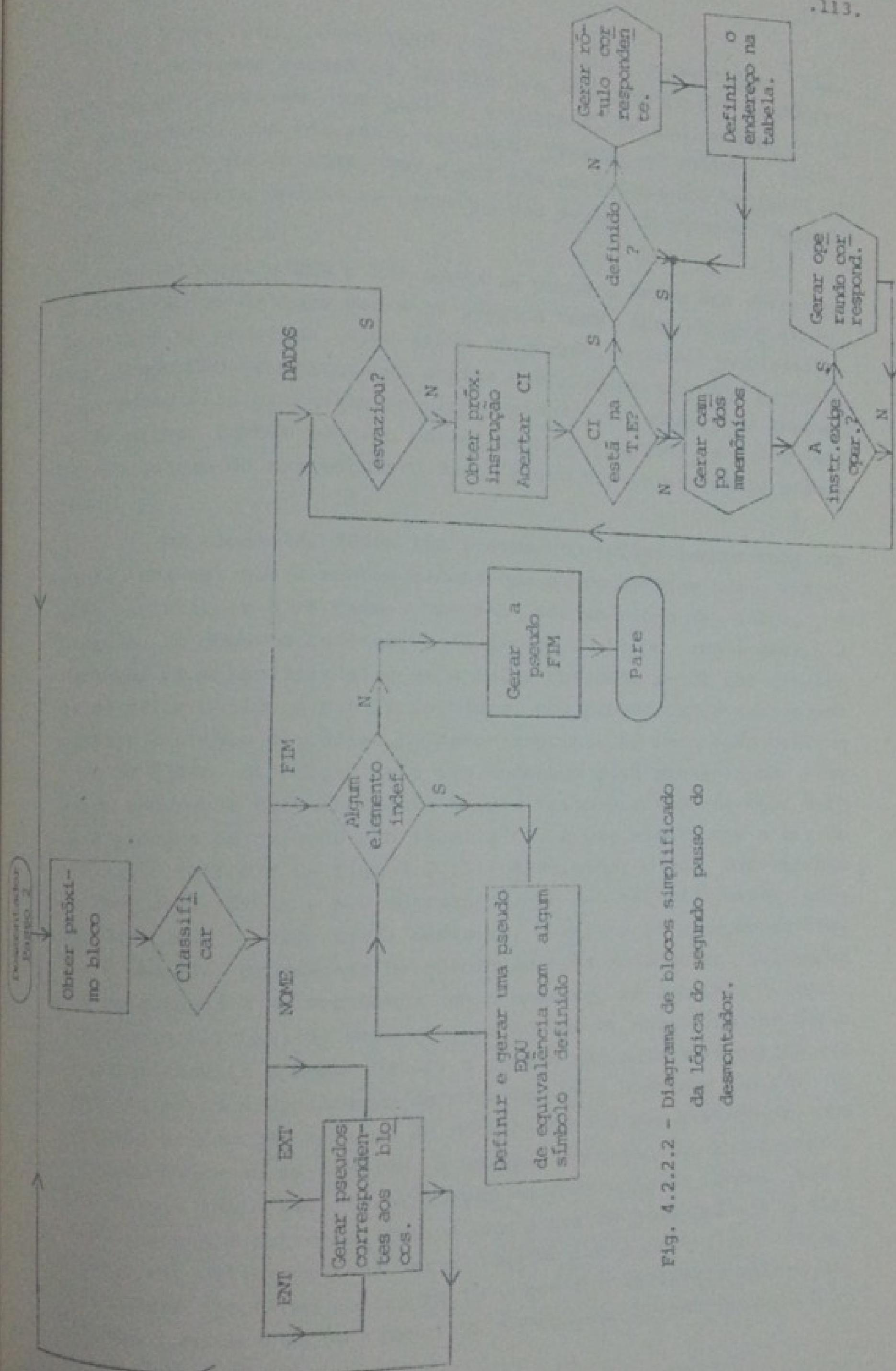



Fig. 4.2.2.2 - Diagrama de blocos simplificado da lógica do segundo passo do desmontador.

Para isto, executado o procedimento mostrado na fig. 4.2.2.2 o programa consta da leitura dos diversos blocos de código objeto, para cada um dos quais é executada, segundo o tipo do bloco em questão, uma diferente rotina de geração de código fonte. Assim, os blocos de NOME, ENT e EXT têm um tratamento bastante simples, que consta apenas de gerações das pseudo instruções adequadas.

Blocos de ENT e EXT geram as pseudos ENT e EXT correspondentes. Blocos de nome geram as pseudos NOME, SUBR ou SEGM conforme o tipo de programa. Além disto, é gerada, quando necessário, uma pseudo COM representando a área comum reservada pelo programa, informação esta existente no bloco de NOME. O endereço de execução do programa, também contido neste bloco, é guardado para ser utilizado quando do aparecimento de um bloco FIM em programas principais.

Os blocos de dados têm tratamento relativamente mais complexo, uma vez que é nestes blocos que se concentram as instruções executáveis e os dados. Uma vez lido um bloco de dados, é extraída do mesmo a informação de origem do bloco, com a qual a variável CI é atualizada, e se o valor corrente de CI for diferente do valor obtido a partir do bloco, uma pseudo ORG é gerada para acertar a origem do código. O passo seguinte é, para cada dado contido no bloco, descobrir se o seu endereço está presente na tabela de endereços referenciados. Isto é feito varrendo-se esta tabela à procura de um endereço igual a CI, e que esteja com o bit de definição (bit D/I da fig. 4.2.1.1) desligado, o que corresponde ao fato de o rótulo a ele correspondente não ter sido ainda gerado. Caso haja sucesso nesta pesquisa, a rotina de geração do rótulos é chamada, produzindo um rótulo de três caracteres alfabéticos relacionados univocamente com o endereço em questão. A seguir, é ligado o bit D/I, de modo que, se houver um novo bloco em que o CI possa assumir o mesmo valor, o rótulo não mais será gerado, evitando-se assim a duplicação de rótulos. Se o endereço não for encontrado na tabela, o campo de rótulos deverá ser preenchido com brancos. Passa-se em seguida, à geração do campo dos mnemônicos. Para isto, é feita uma classificação da instrução de acordo com o grupo a que ele pertence (cap.2), com o que se pode definir o algoritmo a ser utilizado para gerar o mnemônico a partir do código objeto. Este algoritmo lembra muito o que foi utilizado pelo montador para construir o código a partir do mnemônico, apesar de sua fina

lidade ser exatamente a oposta, fato este que é de se esperar se for levada em conta a dualidade da natureza funcional existente entre os programas montador e desmontador. De qualquer maneira, em qualquer caso é possível gerar imediatamente o mnemônico correspondente à instrução nesta fase, faltando apenas o tratamento do campo de operandos. Com base na classificação executada quando da geração do campo de mnemônicos, descobre-se se a instrução em questão exige operando, e, se exigir, de que tipo.

Se a instrução não exigir operando, termina a geração do texto simbólico para a instrução; caso contrário, verifica-se se o operando exigido é uma referência à memória ou uma constante. No caso de o operando ser uma constante, esta é gerada como um número decimal com sinal, sendo impresso no campo de comentários o mesmo número em hexadecimal e em ASCII, se houver correspondente. Se o operando for uma referência à memória, esta referência será gerada como um símbolo de três caracteres alfabéticos, correspondentes ao endereço em questão, devendo ser tomadas, neste ponto, providências para evitar a duplicação de símbolos.

Uma vez gerado o código simbólico, verifica-se se todas as instruções do bloco foram utilizadas. Se isto não acontecer, repete-se todo o processo para a instrução seguinte, até que todo o bloco tenha sido decodificado.

Quando for encontrado um bloco de FIM é varrida a tabela de endereços referenciados, verificando-se se algum dos elementos desta tabela ainda está com o bit de definição desligado, caso em que é gerada uma pseudo EQU, que declara o símbolo correspondente ao endereço em questão como endereço relativo a algum rótulo já definido no programa, no caso de endereço relocável, ou então como endereço absoluto, no caso oposto.

4.3. Alguns detalhes de Implementação

Foram descritos até aqui os aspectos mais gerais do problema da geração de código fonte a partir do código objeto correspondente, independente de considerações relativas às características particulares dos formatos do código objeto ou das instruções de máquina. Serão comentados agora alguns problemas enfrentados durante o objeto dos programas desmontadores implementados, problemas estes decorrentes dos formatos do código objeto e da linguagem de máquina do Patinho Feio.

4.3.1 Desmontador absoluto

Uma fita objeto absoluta carregável pode ser obtida normalmente por um dos três caminhos seguintes:

a) como resultado da montagem de um programa escrito na linguagem do montador absoluto;

b) como resultado da geração de programa objeto absoluto a partir de programas objeto relocáveis, pela utilização do ligador-relocador;

c) como resultado de um "dump" de memória.

E qualquer dos três casos, a fita apresentará as mesmas características estruturais, descritas em A3.1.

Como consequência da simplicidade do formato em que os dados se apresentam numa fita objeto absoluta, decorrem diversos problemas quando se tenta gerar um texto fonte a partir da mesma.

Todos estes problemas têm uma mesma origem, que é a falta de informação relativa aos dados de que é formada a fita objeto absoluta: todos os dados são apresentados da mesma forma, independentemente da sua função no programa. Por isso, é muito difícil construir um programa simples que seja capaz de decidir, em cada ponto do programa objeto absoluto, qual a função específica do dado que está sendo tratado.

Como foi dito em 4.1, optou-se pela simplificação da tarefa, considerando que todo o texto objeto absoluto é formado por instruções. Isto, naturalmente, cria problemas, pois nem sempre todo o código objeto apresentado é formado por instruções, sendo, neste caso, os dados correspondentes a áreas de constantes, interpretados como instruções, e decodificados como tais. Em computadores cujas instruções e cujos dados tenham todos o mesmo comprimento, isto não é realmente grave, uma vez que é sempre possível produzir um programa fonte onde o campo de mnemônicos é preenchido com o resultado da interpretação do dado como instrução, e o campo de comentários, com os diversos formatos correspondentes às constantes cujo valor, em binário, corresponda ao código de máquina em questão. Cabe ao usuário, no caso, escolher a coluna cer

ta da listagem, e, portanto, interpretar a lógica do programa.

Esta não é, no entanto, a situação no presente caso. O Patinho Feio é um computador que apresenta instruções curtas e longas, não sendo portanto possível fazer esta simplificação. Cabe aqui, entretanto, apresentar uma solução parecida com esta, porém mais generalizada, a qual não foi implementada por ter sido julgada dispensável, mas que pode ser utilizada para resolver problemas como este.

Trata-se de gerar, como saída impressa, todas as possibilidades de interpretação de uma sequência de dados. Para computadores com instruções cujo comprimento pode ser de uma ou duas palavras, e dados com uma palavra apenas de comprimento, como é o caso do Patinho Feio, esta solução é viável, uma vez que, além da interpretação como contante, existem, no máximo, duas outras interpretações possíveis: ou o dado corresponde à primeira palavra de uma nova instrução ou então é a segunda palavra de uma eventual instrução longa que começou na palavra anterior. Com três interpretações é possível, portanto, cobrir todas as possibilidades de decodificação da linguagem de máquina de computadores como o Patinho Feio. Para computadores em que o número de formatos de instrução e/ou de dados é maior, o problema se torna cada vez mais complexo, uma vez que o número de interpretações - cresce muito com o número possível de formatos. Para estes casos, apesar de ser possível implementar um programa nestes moldes, as listagens por ele produzidas podem tornar-se demasiadamente extensas e difíceis de acompanhar, o que faz com que sua utilidade seja colocada em dúvida. Um programa como este poderia, no entanto, servir como base para um programa maior e mais complexo, cuja função fosse a de gerar, a partir do programa objeto, das diversas interpretações produzidas, e de eventuais informações adicionais fornecidas pelo usuário, um programa fonte definitivo cuja lógica fosse a mais próxima possível da do programa decodificado. Trata-se portanto de um programa rudimentar de inteligência artificial, que não cabe neste contexto.

Um outro problema, também causado pela existência de instruções de comprimentos diferentes, surge quando o programa a ser decodificado é resultado de um "dump" de memória, onde é frequente o aparecimento de blocos de dados em que o último dado é a primeira palavra de uma instrução, que tem sequência no bloco seguinte. Programas objeto gerados pelo montador ou pelo ligador-

-relocador não apresentam este incoveniente, uma vez que estes programas geram blocos de código objeto com um número inteiro de instruções. Como, no entanto, a maior utilidade do desmontador absoluto é a da decodificação de "dumps" de memória, foi incluída na sua lógica a deteção de casos em que o último dado do bloco pode ser interpretado como a primeira palavra de uma instrução longa, caso em que é lido o próximo bloco e testada sua origem. Se esta corresponder à posição de memória seguinte à ocupada pela última palavra do bloco anterior, o desmontador considerará os dois blocos de dados como logicamente adjacentes, gerando a instrução longa correspondente. Caso contrário, é gerada uma pseudo DEFC correspondente à última palavra do bloco anterior, e uma pseudo ORG definindo a nova origem para o bloco autal.

Outro problema surge, com menor frequência, durante a fase de decodificação. Trata-se dos casos em que o dado a ser decodificado não corresponde a nenhuma instrução válida. Nesta situação, optou-se pela geração de uma pseudo DEFC, tratando-se normalmente os dados seguintes.

Isto ocorre em áreas de dados, onde é muito provável o aparecimento de constantes que não correspondem a instrução alguma . Consequentemente, se uma área de dados estiver intercalada entre duas áreas de programa, é possível, que,devido à ocorrência de uma situação como esta, aconteça de a primeira palavra da área de programa que segue a área de dados ser interpretada como parte de uma instrução longa iniciada na última palavra de área de dados. O resultado é uma defasagem do contador de instruções, do que decorre uma interpretação não exata do programa objeto.

4.3.2 Desmontador Relocável

Em um programa objeto relocável do Patinho Feio, os dados correspondentes a constantes são marcados como tais pelo montador, com o que os dois grandes problemas enfrentados na implementação do desmontador absoluto simplesmente deixam de existir.

Surgem, no entanto, outros problemas, relacionados desta vez com o caráter relocável do programa objeto que se deseja decodificar.

Como foi visto no cap. 2, os programas relocáveis apresentam, no bloco de NOME, um indicador do comprimento da área comum de dados. No programa fonte gerado, é necessário atribuir um nome a esta área comum, para permitir referências às mesma. Por outro lado, alguns nomes, como os de pontos de entrada e nomes de áreas globais (declaradas nas pseudos ENT e EXT), são pré-estabelecidos, não sendo permitida a sua utilização pelo desmontador na geração automática de rótulos a partir do endereço. Aparece assim o problema da escolha de rótulos tais que não sejam os mesmos utilizados para as declarações de pseudos ENT e EXT, e que sejam diferentes do utilizado como nome da área comum. Para solucionar este problema, o primeiro passo, ao construir a tabela de endereços referenciados, guarda também os nomes dos rótulos utilizados para estas pseudos. Durante a geração dos rótulos, no segundo passo, é consultada a tabela dos rótulos já utilizados, evitando-se desta maneira a utilização do mesmo rótulo mais de uma vez.

Devido à existência de endereços de tipos diferentes (identificados no texto objeto pelos indicadores de relocação), podendo tais endereços coincidir uns com os outros, torna-se necessário utilizar, na geração dos rótulos a ele correspondentes, algoritmos diferentes. A opção adotada na implementação do desmontador do Patinho Feio foi a seguinte:

- a área comum consta de um bloco único com um só nome, e todas as referências aos dados da mesma são relativas ao primeiro elemento, cujo nome é o do bloco.

- os pontos de entrada e os nomes globais externos são mantidos inalterados, e obtido dos blocos ENT e EXT do texto objeto.

- as referência absolutas não geram rótulos, sendo efetuadas diretamente, em hexadecimal.

- as referências relocáveis geram rótulos segundo um algoritmo de geração que transforma os endereços em seqüências de três caracteres.

- por meio de pesquisa em uma tabela de símbolos,evita-se a duplicação de símbolos no programa.

O tratamento do bloco de FIM do programa objeto relocável é análogo ao tratamento do final de fita do programa absoluto: a tabela de endereços referenciados é consultada,gerando-se pseudos EQU correspondentes aos endereços indefinidos. A diferença principal entre os dois desmontadores, no que se refere a este aspecto, reside no fato de o desmontador absoluto gerar equivalência com posições absoluta de memória, ao passo que, no caso relocável, esta equivalência se faz com um endereço relativo ao início da área comum, ou então ao início do programa dependendo da informação de relocação associada ao endereço em questão.

4.4 Conclusões

Os desmontadores têm se revelado de grande utilidade no desenvolvimento de programas para o Patinho Feio. Apresentam, no entanto, algumas deficiências originadas principalemente na falta de informações apresentada pelos programas objeto que se deseja desmontar. É possível construir desmontadores mais eficientes e completos desde que se introduzam algumas modificações nos programas que geram as fitas objeto. Assim, é possível, por exemplo, incluir opcionalmente, nas fitas objeto geradas por programas - montadores, compiladores ou relocadores, informações adicionais sobre os nomes das diversas variáveis, sobre o tipo de instrução ou pseudo instrução que deu origem a cada determinado código, e assim por diante. Evidentemente, isto não é prático depois que o sistema já está desenvolvido, mas pode ser previsto quando da fase de projeto dos diversos módulos de "software" básico do computador, uma vez que as adaptações de programas prontos geralmente - levam a soluções difíceis e deselegantes, o que não ocorre quando tais considerações são feitas na ocasião do projeto dos vários módulos.

Como aplicação principal, pode-se citar a ajuda na decodificação de programas objeto gerados por programas tradutores e ligadores em fase de teste. Quanto à sua utilização como ajuda na

recuperação e documentação de programas, deve-se levar em conta que às vêzes é muito mais racional refazer um programa que simplesmente tentar compreender a lógica de um programa não documentado. Recomenda-se portanto a utilização dos desmontadores para esta finalidade apenas em casos em que isto é indispensável, com o que é possível que se economize muito tempo e mão de obra.

Levando-se em conta as restrições do desmontador absoluto no que se refere à presença de constantes entre duas áreas de programa, é aconselhável que, durante a fase de codificação dos programas, seja este fato levado em consideração, agrupando-se ao máximo as constantes e áreas de trabalho, sugerindo-se que a área assim obtida seja finalizada por uma instrução curta para que,caso o desmontador erre na interpretação das instruções, possa ser recuperado o controle ao se iniciar a decodificação da área de programa seguinte. Deve-se observar que, apesar de esta condição ocorrer apenas em programas absolutos, é recomendável que seja tomado este cuidado também em programas relocáveis se se desejar garantir que o programa será bem interpretado mesmo após sua manipulação pelo ligador-relocador.

CAPÍTULO 5. A ROTINA DE DEPURAÇÃO DE PROGRAMAS

3. A ROTINA DE DEPURAÇÃO DE PROGRAMAS

3.1 - A fase de depuração de um programa

Um dos mais difíceis trabalhos enfrentados por um programador é o da deteção, dignóstico e correção de erros de lógica em seus programas. Isto se deve ao fato de que tais erros são devidos às mais diversas e imprevisíveis causas tornando sua deteção uma tarefa demorada e cansativa, principalmente pelo fato de ser às vezes necessário para isto acompanhar passo a passo o resultado das diversas operações intermediárias executadas pelo programa, desde seu início.

Quando não se dispõe de recursos em "hardware" que facilitem este trabalho, (como por exemplo, a geração automática de um pedido de interrupção ao fim da execução de cada instrução),o processo normalmente empregado, é o do acompanhamento do programa, instrução por instrução, no painel do computador. Este processo tem inúmeras desvantagens sendo as mais importantes as seguintes:

a) é excessivamente lento e sujeito a erros de leitura do painel, por parte do operador;

b) não fornece uma cópia impressa do que está ocorrendo, para consulta futura. Os eventos devem ser portanto, detetados no instante de sua ocorrência,não sendo possível a recuperação das informações "a posteriori";

c) desejando-se conhecer o conteúdo da memória a qualquer momento, é necessário interromper o processo, em andamento, de execução do programa, devendo-se, para continuá-lo, restaurar manualmente o conteúdo de todos os registradores alterados durante a exposição ou modificação do conteúdo da memória;

d) exige operação frequente dos controles do painel, sendo por isso bastante sensível a falhas de operação ou a defeitos nos controles.

Apesar destes incovenientes, este método tem uma vantagem muito grande sobre qualquer outro: exigindo o acompanhamento do programa passo a passo, permite corrigir o erro assim que este for detetado, possibilitando que se impeça que os seus efeitos sejam propagados, permitindo assim que mais de um erro possa ser detetado em um único teste, sem a necessidade de recarregar o programa. Assim, ainda é às vezes conveniente depurar programas pelo painel, apesar de todas as desvantagens citadas, especialmente quando se sabe a"priori"que o erro está localizado em um trecho restrito do programa.

Como, no entanto, este não é normalmente o caso, e como o computador utilizado não dispõe de recursos de "hardware" para facilitar este trabalho, um meio de tornar este processo mais rápido e menos sujeito a erros é, utilizando o próprio Patinho Feio, construir um programa que supervisione a execução do programa que se deseja depurar, e que, além disto, forneça opcionalmente uma listagem ("trace") de todos os registradores, estado da máquina, endereço efetivo de instruções de referência à memória, para cada instrução executada, "dumps" de memória, etc.

5.2 A filosofia da rotina de depuração

O objetivo principal de uma rotina de depuração, é a de fornecer informações a respeito do andamento do programa.Pode-se desejar, em algumas ocasiões, apenas informações sobre o conteúdo dos registradores no instante em que uma subrotina é chamada. Em outros casos, porém, é interessante que se possa dispor, por exemplo, da evolução completa da execução de uma subrotina. Em outros, ainda, poder-se-á desejar interromper o processamento e saber o estado dos registradores assim que for executada uma determinada instrução, ou então quando uma determinada posição de memória for referenciada.

Numerosas opções existem, para a execução dessa tarefa, cada qual apresentando suas vantagens e desvantagens. Naturalmente, o programador gostaria de poder dispor de todas simultaneamente, porém, no caso de um minicomputador do porte do Patinho Feio, isto acarretaria um uso excessivo de memória, com o que, pouco

espaço sobraria para o programa a depurar, não sendo portanto viável a depuração, na máquina, de programas grandes com este recurso. Se se dispuser de um computador maior, pode-se contornar o problema escrevendo, para executar em tal computador, um simulador-interpretador da linguagem de máquina do minicomputador. Com ele podem ser executados programas do minicomputador com a possibilidade de utilização de todos os recursos disponíveis no computador hospedeiro, além de ser possível, a qualquer momento, ligar as opções de "trace", "dump", etc. (cap.3).

É claro que a utilização do simulador só é indispensável em caso extremos, como quando a deteção de um erro no próprio computador se tornar demasiadamente trabalhosa ou demorada, ou então quando o programa a depurar e a rotina de depuração não couberem juntas na memória.

Baseados em todas estas considerações, pode-se estabelecer características desejáveis para uma rotina de depuração:

a) deve ser compacta para não ocupar excessivamente a memória;

b) deve ser suficientemente versátil para permitir ao usuário o uso da maioria dos recursos citados;

c) não deve restringir a utilização de nenhuma característica da máquina;

d) deve ser de fácil utilização;

e) deve ser capaz de fornecer o maior número possível de informações sobre o andamento do programa em depuração;

f) não deve ser excessivamente lenta.

A depuração de um programa, quando comandada por rotinas deste tipo, pode ser executada de diversas maneiras, cada uma das quais apresentando vantagens e desvantagens peculiares. Destacam-se a seguir 3 opções:

1. <u>A rotina de Acompanhamento é construída pelo pró-</u>

<u>prio usuário.</u> Neste caso, o usuário pode incluir a saída de todas as informações desejadas, por exemplo a análise da evolução do conteúdo de registradores ou de posições de memória, em observação, podendo tecer, a partir de tal análise, possíveis diagnósticos da falha pesquisada. Naturalmente uma rotina deste tipo, apesar de ser a mais eficiente para acompanhar a lógica de um particular programa nem sempre poderá ser utilizada para isto em outros programas,devido ao seu caráter específico, sendo portanto usada apenas quando for necessário um controle rigoroso da evolução de eventos particulares do programa em questão. Assim , tais rotinas são construídas pelo próprio programador, e são executadas em posições estratégicas do programa em observação, sendo portanto necessário, para sua incorporação, compilar ou montar novamente todo o programa. Ao se escrever uma rotina dessas, é conveniente que se incluam alguns testes que permitam ligar ou desligar a saída dos diversos relatórios que tais rotinas possam fornecer. Isto é efetuado normalmente através da leitura das chaves do painel. Uma rotina simples, mas bastante útil em inúmeras situações, pode por exemplo imprimir uma identificação de cada subrotina chamada durante a execução do programa, e, opcionalmente, o conteúdo dos principais registradores e variáveis do programa nesta ocasião. Neste caso, é necessário acrescentar como primeira instrução executável de cada subrotina, uma chamada da rotina de acompanhamento, a qual deverá naturalmente salvar todos os registradores que irá utilizar, restaurando-os antes de retornar à rotina que a chamou.

2. <u>A rotina de depuração é uma subrotina do sistema.</u> Deve ser montada juntamente com o programa a ser depurado. Neste caso, a execução do programa a depurar será comandada por um pequeno monitor, cuja função é a de aceitar comandos do operador, os quais comandam a geração de relatórios que devem conter as informações requisitadas. Por meio de tais comandos pode-se solicitar "dumps" de memória, modificar posições de memória ou registradores, estabelecer pontos onde se deseja que

a execução do programa seja interrompida, etc. Sempre que o número de pontos de observação do andamento do programa não for muito grande, é prático que o monitor substitua as instruções neles existentes por desvios para a rotina de relatório, que, uma vez executada, retorna para o monitor, o qual passa a aguardar um novo comando. Como se pode notar, o monitor só terá o controle quando o programa passar pelos pontos de observação, sendo portanto sua função principal, durante a execução do programa, substituir as instruções contidas nos pontos de observação por desvios para as rotinas de relatório e promover a execução das instruções deles retiradas quando for recebido um comando de execução.

A vantagem de um procedimento como este é clara: a velocidade de execução do programa não é alterada substancialmente pela ação da rotina de depuração, a qual pode ser bastante simples, e portanto compacta. Sua principal desvantagem é a de não ser prática no caso de se desejar estabelecer regiões e não pontos de observação.

3. <u>A rotina de depuração é um simulador-interpretador da linguagem de máquina do computador</u>. Neste caso, o controle está permanentemente com a rotina de depuração para a qual as instruções do programa em teste são apenas uma sequência de dados que por sua vez são considerados como comandos pelo interpretador. A análise de tais comandos permite que o interpretador execute uma subrotina conveniente em cada caso, simulando assim em nível de registradores a execução real do programa em questão. Uma vantagem do processo é que se pode manter, passo a passo, um controle completo do que está acontecendo, uma vez que tudo acontece sob o comando do programa interpretador. Isto permite que se acionem rotinas de relatório cada vez que acontecer uma referência a alguma posição da memória pertencente a uma certa área, e também fornece a possibilidade de um acompanhamento completo da execução de cada instrução do programa cada vez que este passar por regiões espe

cificadas de memória. Como se pode notar, a velocidade de execução do programa fica bastante prejudicada, o que torna o processo incoveniente para a depuração de determinados programas como por exemplo os que envolvem aquisição rápida de dados. Entretanto, como a maioria dos programas não são tão dependentes da velocidade de execução, as vantagens descritas tornam muito indicada esta opção para a grande maioria dos programas normalmente utilizados, fato que foi decisivo na escolha deste método para a implementação do exemplo aqui descrito.

5.3 Os problemas enfrentados

Adotada a opção de construir uma rotina de depuração que seja um simulador-interpretador, deve-se enfrentar praticamente todos os problemas apresentados por uma tarefa de emulação. Para o caso em que a máquina hospedeira é a própria máquina emulada, o trabalho de simular a execução da maioria das instruções é bastante facilitado, não sendo necessário construir uma rotina para cada instrução a simular (capítulo 3), mas sim uma para cada grupo de instruções de funcionamento semelhante

Um dos problemas mais sérios enfrentados tanto na simulação de programas num interpretador como na emulação de um sistema em outro é o da compatibilização da entrada e saída da máquina emulada com a do sistema hospedeiro. Isto se deve ao fato de os dois sistemas normalmente apresentarem a mesma lógica de interrupção, sendo na maioria dos casos bastante difícil implementar uma simulação perfeita do sistema de entrada e saída sem alterar o "hardware" da máquina hospedeira. A situação á agravada ainda mais porque obviamente a máquina hospedeira deve utilizar alguns de seus periféricos para seu próprio uso e outros para substituirem os da máquina emulada. Isto implica em compartilhar o sistema de interrupção da máquina hospedeira com a "máquina" - emulada, o que quer dizer que as rotinas de tratamento de interrupção da máquina hospedeira serão utilizadas por ambas.Como foi visto no capítulo 3, quando se simula uma máquina em outra constrói-se um modelo matemático para o circuito de entrada e saída, e trabalha-se sobre este modelo gerando, por programa, alterações no estado do mesmo em ocasiões oportunas, e usando rotinas de

E/S do sistema para executar as operações de entrada e de saída propriamente ditas em instantes convenientes. Ora, isto nem sempre pode ser feito quando a máquina simulada e a hospedeira são a mesma pois neste caso a rotina de interrupção poderia eventualmente ser a própria rotina a depurar, e portanto estar sujeita a falhas. Outro fato merece ser realçado neste caso: sempre que ocorrer uma interrupção, o programa de depuração perderá o controle, o qual passará para a rotina de tratamento de interrupção, e isto faz com que o acompanhamento da rotina de interrupção não seja possível. Se o programa a depurar for a própria rotina de interrupção, então a rotina de depuração não terá utilidade. Este tipo de problema pode ser solucionado adotando--se um esquema híbrido das opções 2 e 3 apresentadas em 5.2. A opção 2 seria aplicada apenas para a rotina de tratamento de interrupção: assim que ocorrer uma interrupção, a rotina de tratamento, devidamente alterada pelo monitor da rotina de depuração, desviará o processamento para uma rotina de acompanhamento de interrupções, que seria parte do monitor. Tratada a interrupção, o monitor devolveria o controle ao programa interrompido podendo este continuar a simulação em andamento. Esta solução apresenta a desvantagem de não se aplicar a todos os periféricos , pois os que são compartilhados pelos sistema hospedeiro e simulado apresentarão conflitos na ocasião do tratamento de suas interrupções.

Como se pode notar, apesar de haver soluções para o problema, tais soluções são trabalhosas e não totalmente satisfatórias, podendo ser melhoradas parcialmente inibindo o sistema de interrupção durante a execução de rotina de depuração, e liberando-o em ocasiões oportunas. Devido à complexidade de tais soluções e das restrições a que estão sujeitas, resolveu-se restringir o tipo de programas a depurar ao invés de elaborar uma rotina de depuração geral complexa e portanto extensa. Sendo o sistema de interrupção a maior fonte de problemas, impôs-se que o programa a depurar deva ser executado com o sistema de interrupção inibido. O tratamento de entrada e saída torna-se assim muito simples, bastando simular as instruções de entrada e de saída de dados propriamente ditas, ignorando todas as instruções de controle de entrada e saída. Naturalmente, a execução correta do programa sob o comando da rotina de depuração não implica que

o mesmo esteja completamente correto, podendo-se apenas afirmar, neste caso, que o programa está correto a menos das rotinas de entrada e saída. Esta restrição não é forte desde que o usuário utilize rotinas de entrada e saída do sistema, uma vez que estas devem, por hipótese, estar corretas. Para utilizar a rotina de depuração, o usuário deverá portanto substituir todas as chamadas de rotinas de entrada e saída por chamadas de rotinas equivalentes que não utilizam o sistema de interrupção, e, uma vez depurada a lógica do programa, restaurar as chamadas das rotinas substituídas. Técnicas mais gerais de simulação de entrada e saída foram vistas no capítulo 3, e sua implantação em rotinas de depuração como esta só seria prática se o sistema fosse de porte substancialmente maior.

5.4 Exemplo de Implementação

Com base nas considerações expostas anteriormente, foi elaborada para o Patinho Feio uma rotina que é basicamente um interpretador da linguagem de máquina do mesmo, e que simula a execução do programa a depurar, fornecendo um "trace" do programa em uma região especificada da memória. Devido à necessidade de manter livre o máximo possível da memória do computador, resolveu-se restringir todas as facilidades supérfluas, elaborando-se, portanto, apenas as rotinas de entrada de dados, de interpretação da linguagem de máquina, e de relatório. Na fig. 5.4.1 está representado um fluxograma simplificado da lógica do exemplo implementado.

5.4.1 - Rotina de Entrada de Dados

Consta apenas de algumas instruções de leitura de painel e de atribuição de valores iniciais a alguns parâmetros, para uso do interpretador. Os dados fornecidos pelo painel são: o endereço do início lógico do programa, e os endereços inicial e final da região onde se deseja que o simulador forneça o "trace". Em uma versão mais sofisticada, esta rotina poderia ser substituída por um interpretador de comandos fornecidos pela console, que per

mitissem ao operador solicitar ao programa:

a) "dumps" de memória;

b) parada de processamento (ao passar por regiões especificadas de memória);

c) parada de processamento quando o programa referenciar uma posição de memória pertencente a alguma região - especificada;

d) cancelamento ou modificação de qualquer uma das opções solicitadas;

e) modificação do conteúdo de alguma posição de memória;

f) modificação do conteúdo de algum registrador;

g) listagem do conteúdo dos registradores, etc.

Como se pode notar, apesar de a lógica de tal interpretador ser bastante simples, o volume de informações a serem tratadas e o número de opções desejáveis são muito grandes, o que faz com que a quantidade de memória necessária para sua implementação seja muito grande em comparação com a utilizada pelo próprio interpretador. Sua inclusão numa versão sem armazenamento secundário torna-se por isso inviável uma vez que não sobraria espaço para o programa a depurar. Por outro lado, em uma configuração com disco torna-se bastante simples criar uma área de "overlay" compartilhada pelas diversas rotinas que compõem este interpretador de comandos. Como um único comando é processável de cada vez, apenas a rotina correspondente ficaria residente na memória, com o que seria possível a inclusão das diversas opções acima enumeradas sem prejuízo de área de memória. Quanto ao tempo extra consumido pelos acessos ao disco, deve-se levar em conta que, toda vez que um comando é fornecido, o tempo gasto pelo operador para escrever o comando na console ou pela máquina para fornecer o relatório pedido é várias ordens de grandeza maior que o tempo de acesso ao segmento do programa residente em disco, com o que tal excesso de tempo será totalmente desprezível para todos os efeitos se os segmentos forem projetados convenientemente.

5.4.2 Rotina de Interpretação e Execução da Linguagem de Máquina.

A função principal da rotina de entrada dos dados é fornecer à rotina de execução o endereço a partir do qual o programa deve ser executado. Assim, o endereço de partida ou contador de instruções (CI) deve estar já preparado no instante em que o comando passar à rotina de execução. Esta rotina executa em primeiro lugar a busca e decodificação da instrução apontada pelo CI, passando em seguida à simulação da execução da mesma. Como a linguagem interpretada é a própria linguagem de máquina do computador hospedeiro, decorrem deste fato consideráveis simplificações na simulação, uma vez que em grande número de casos o próprio "hardware" da máquina pode executar a instrução em questão sem que o programa simulador perca o controle. Com efeito, somente as instruções que envolvem quebras de sequência (pulos ou saltos condicionais) deverão ser realmente simulados, uma vez que, se a sua execução for efetuada diretamente pelo "hardware", o registrador contador de instruções da máquina será modificado pela execução da instrução, sendo portanto desviado o processamento para o programa simulado, perdendo pois o interpretador o controle da operação.

O mesmo deverá acontecer com instruções que atuam sobre o sistema de interrupção do computador, especialmente as instruções do grupo de entrada e saída. Como foi explicado, a simulação é executada com o sistema de interrupção inibido, e portanto a execução de instruções como PERM (permite interrupção) ou de instruções que de um modo ou de outro criem situações favoráveis ao aparecimento de pedidos de interrupção deverá ser evitada ou substituída por ações que, externamente, sejam equivalentes. Isto é feito, no caso aqui descrito, por meio de testes simples que fazem com que a maior parte das instruções do grupo de entrada e saída, bem como PERM, não sejam executadas. Deste grupo são simuladas apenas as instruções de "ENTR", "SAI" e "SALTO DE ESTADO ESTIVER LIGADO". As duas primeiras são substituídas por pulos para subrotinas de entrada e de saída de dados no canal correspondente, e a terceira por um salto incondicional. Todas as demais funções de entrada e saída são ignoradas, funcionando na simulação como instruções de desvio incondicional para a instrução seguinte.

A terceira função da rotina de execução é a de acertar o valor do CI para que este, após a execução da instrução, aponte corretamente a próxima instrução a executar.

5.4.3 Rotina de Relatório

Esta rotina tem por finalidade imprimir, quando necessário, o relatório conveniente, o qual será utilizado pelo operador para a deteção e diagnóstico dos possíveis erros de lógica do programa.

Um relatório para tais fins deverá ser o mais completo possível para que o programador tenha dados suficientes para detetar os erros do programa. Esse relatório deve ser entretanto suficientemente conciso para que sua impressão seja rápida e que o volume de informações não seja exageradamente grande, o que seria um desperdício de tempo e de espaço. Tendo em mente estas considerações, escolheu-se o seguinte formato de relatório: para cada instrução é impressa uma linha, que inclui o valor do CI, o código de máquina da instrução, o valor do conteúdo de todos os registradores, e, se a instrução referenciar a memória, o endereço efetivo da referência, e os conteúdos da posição referenciada e da posição seguinte. (fig. 5.4.3.1).

(a)

CI	INSTRUÇÃO	ACUMULADOR	EXTENSÃO	ÍNDICE	T,V

(b)

CI	INSTRUÇÃO	ACUMULADOR	EXTENSÃO	ÍNDICE	T,V	END.EFETIVO	CONTEÚDO DO END. EFET. E DA POSIÇÃO SEGUINTE

(a) - Para instruções que não referenciam a memória.

(b) - Para instruções de referência à memória.

Fig. 5.4.3.1 - Formato de saída do relatório da rotina de depuração

A rotina de relatório apresenta, na versão mais compacta do programa de depuração, apenas o código, em notação hexadecimal, de instrução executada, no campo "Instrução". Na versão mais completa, é impresso também o mnemônico correspondente à instrução.

Inicialmente, a rotina de impressão de relatório executa um teste para verificar se o valor do CI está compreendido entre os limites fornecidos para o "trace". Em caso afirmativo, o relatório é impresso, sendo para isto consultadas as variáveis utilizadas pelas rotinas que salvam e restauram os registradores da máquina, as quais contêm cópias dos diversos registradores em questão. Para o caso das instruções de referência à memória, o cálculo do endereço efetivo do operando é efetuado e também o resultado deste cálculo é impresso, bem como os valores do conteúdo desta posição de memória e da seguinte.

Nas versões implementadas, não foram incluídas paradas de processamento devidas a referências a uma área de memória (descrita como "Proteção de Memória" em 3.6). A rotina de relatório deverá ser chamada sempre que ocorrer uma parada deste tipo ou de qualquer outro que venha a ser implantado no programa. Para tanto, deverá ser executado um teste de condição de impressão de relatório também durante o cálculo do endereço efetivo, bem como após a execução de cada instrução, com inevitável queda de velocidade do programa de depuração.

Cabe frisar também que sempre que a mesma rotina de relatório deva servir a diversas finalidades, como é o caso aqui sugerido; a causa da impressão do relatório deverá acompanhar o mesmo. Isto é facilmente executado passando-se, como parâmetro, para a rotina de relatório, a causa da solicitação de impressão do mesmo. As principais causas de uma solicitação de relatório são as seguintes:

a) a instrução corrente está dentro dos limites do "trace" escolhido;

b) a instrução corrente está em um ponto de parada ("breakpoint") escolhido pelo operador;

c) a instrução corrente referencia uma região especificada;

d) a instrução corrente tentou modificar uma região especificada;

e) a instrução corrente está fora dos limites permitidos pelo programador;

f) a instrução corrente é inválida;

g) a instrução corrente poderia tirar do programa de depuração o controle do processo (por exemplo, é uma instrução que manipula o sistema de interrupção do computador);

h) a instrução corrente apresenta um número escessivo de níveis de endereçamento indireto (o número máximo poderia ser fornecido pelo programador).

De todas estas opções, está implantada na versão mais compacta do programa de depuração apenas a primeira. Na versão completa, pode-se contar com as opções a) e b).

5.4.4 - Comentários e Observações

- Como foi sugerido em 5.4.3, há atualmente duas versões do programa de depuração.

• A primeira ocupa cêrca de 9ØØ palavras, e é a mais compacta, destinando-se à depuração de programas de até 3K palavras. Não apresenta nenhuma facilidade ou recurso adicional permitindo apenas a execução de "traces" em regiões especificadas. No relatório, as instruções são impressas em notação hexadecimal apenas.

• A segunda ocupa cêrca de 2ØØØ palavras, sendo recomendável para programas menores, permitindo a solicitação de pontos de parada ("breakpoints"), para facilitar a operação de depuração. Além disto, inclui na rotina de relatório uma decodificação dos códigos de instrução, imprimindo os mnemônicos ao lado dos códigos de máquina das instruções. Adicionalmente, permite

solicitar, pelo painel de chaves, a geração de fitas objeto car regáveis, "dumps" de memória em notação hexadecimal, modificação de posições de memória, etc. Estas facilidades adicionais tiveram como implicação principal o consumo de memória, não se refletindo porém na velocidade do processamento.

- O programa de depuração revelou-se, apesar de sua grande simplicidade, bastante útil na deteção de erros de programação, quando utilizado com critério. Evidentemente, para programas muito grandes (com mais de 3K palavras), bem como para programas que utilizam o sistema de interrupção, as facilidades do simulador-interpretador do capítulo 3 tornam este último mais indicado que o programa de depuração aqui apresentado.

- A expansão do programa de depuração com a finalidade de implementar outras facilidades de utilização e outros recursos para a depuração será viável em uma configuração com disco, a qual poderá comportar um simulador-interpretador do porte do descrito no capítulo 3, valendo então todas as considerações tecidas em relação ao mesmo, e com a vantagem de apresentar velocidade substancialmente mais alta devido à não necessidade de simular a maioria das instruções.

CAPÍTULO 6. O EDITOR SIMBÓLICO

6. O EDITOR SIMBÓLICO

6.1 - Introdução

Quando se dispõe de um computador cuja unidade principal de entrada é uma leitora de fita perfurada, defronta-se infalivelmente com a desagradável e frequente tarefa de alterar as informações contidas em fitas de papel.

Do ponto de vista do usuário, sendo a fita perfurada um arquivo de dados de acesso sequencial (ao contrário de um arquivo formado por um conjunto de cartões perfurados, onde o acesso é aleatório), sua alteração exige a perfuração de um novo arquivo, após o que o antigo se torna obsoleto, devendo ser portanto inutilizada a fita que o contém. Este processo de correção, quando executado manualmente é relativamente trabalhoso sempre que o arquivo a ser alterado for extenso, exigindo por parte do operador uma atenção muito grande para que a correção seja executada com perfeição (Isto não acontece com a alteração dos dados contidos em arquivos de cartões perfurados, que é trivial pela facilidade de manuseio do mesmo). Assim sendo, existe uma conveniência muito grande de se tornar mais simples e automático o processo de alteração de um arquivo em fita. Aos programas que executam esta tarefa dá-se o nome de Editores Simbólicos. Estes programas podem ser utilizados não apenas para a modificação de arquivos em fita perfurada mas também para qualquer tipo de arquivo com estrutura semelhante,podendo-se utilizá-los,por exemplo,para a manipulação de arquivos sequenciais em discos, fitas magnéticas, etc.

Em qualquer caso, a existência de unidades de memória de massa no sistema é extremamente útil não sendo, entretanto,imprescindível.

Num caso geral, um editor simbólico deve ser adaptável à configuração do computador no instante da utilização. Assim, é conveniente que o operador tenha a possibilidade de escolher as unidades de entrada por onde serão fornecidos o arquivo original, o programa de modificação e o conjunto de modificações, bem como a unidade de saída onde o arquivo devidamente alterado deverá ser produzido. Usualmente o "programa de modificação" (sequência de instruções para o editor) e as modificações propriamente ditas (sequência de dados a serem usados pelo editor para alterar o arquivo original) são fornecidas na mesma unidade de entrada.

No presente caso, dispõe-se de uma unidade de leitura rápida de fita perfurada e duas "teletypes", uma das quais é utilizada como console (unidade de entrada do programa de modificações e das correções) e outra como unidade de saída (de listagens e da fita modificada). A leitora rápida é a unidade de entrada do arquivo original a modificar. Há portanto a necessidade de utilizar algum símbolo de identificação que permita ao editor reconhecer se uma certa linha lida na console é uma instrução ou um dado.

Para isso, convencionou-se que todas as instruções devem ser precedidas de um símbolo especial (no caso, foi utilizada uma barra ("/")).Com isto, os dados ficam limitados: não pode ocorrer barra no início de uma linha de dados, pois isto, por convenção, significa que a linha é de instrução. Para contornar esta limitação, convencionou-se que uma linha iniciada por uma sequência de duas barras é uma linha de dados, onde a primeira barra deve ser ignorada.Torna-se assim possível a existência de linhas de dados iniciadas por barra, com o que é viável a modificação de quaisquer arquivos, sem o risco de ambiguidade entre instruções e dados.

6.2 - A filosofia do Editor

É bastante grande o número de opções que se pode efetuar na fase de projeto de um editor simbólico. Por isso, é necessário levar-se em conta diversos fatores que podem influir na escolha de sua arquitetura, entre os quais podem ser destacados os seguintes:

a) Finalidade a que se prestará o editor. É com base neste fator que se deve projetar o conjunto de instruções do editor, incluindo ou não determinados tipos de comando, dependendo da aplicação dos mesmos para a finalidade a que se destina o editor.

b) Facilidade de Edição. Com a intenção de facilitar a operação do editor, é importante que o conjunto de instruções seja composto de comandos ao mesmo tempo poderosos e de fácil memorização.Assim, é importante que a sintaxe dos comandos seja a mais simples possível para que o operador não cometa enganos devidos à excessiva complexidade dos mesmos. Por outro lado, os comandos

deverão ser suficientemente poderosos para que o usuário não seja obrigado a executar um número exagerado de comandos simples para atingir seu objetivo, especialmente para as operações mais frequentes.

c) Configuração disponível. Um fator importantíssimo para o projeto do conjunto de instruções de um editor é a configuração do computador em que ele deverá ser utilizado. Isto se deve especialmente ao fato de que a existência ou não de um determinado periférico ou a capacidade de memória disponível são fatores que podem ou permitir ou então inviabilizar a implementação de determinados comandos. A influência de tais fatores também se faz sentir na eficiência do editor no que se refere a instruções que podem ser implementadas independentemente da existência de um dado recurso de "hardware", mas que são mais adequadas ao uso de tal recurso. Pode-se citar, como exemplo de uma situação como esta, um comando de comparação de dois arquivos. É possível implementá-lo sem o uso de memória de massa, porém o tempo necessário para sua execução será exageradamente maior, sendo além disto muito menos confiável tal operação neste caso.

d) Velocidade de Execução. O editor deve ser projetado de tal maneira que a sua execução seja a mais rápida possível. Deve-se lançar mão, para isso, de todos os periféricos rápidos disponíveis, pois a maior parte do tempo de edição é consumida na execução de cópias de trechos de arquivos. Além desta otimização trivial do tempo de edição, é possível, implementar, mediante um projeto criterioso dos formatos dos arquivos, uma estrutura de informações eficiente que permita, numa versão com memória de massa, uma edição rápida sem necessidade de cópia total do arquivo (ref. 12).

6.2.1 - Definição do conjunto de instruções do editor

Com base nas considerações anteriores, foram tomadas as seguintes decisões quanto ao conjunto de instruções que o editor deveria comportar:

a) Como a maioria das operações de edição correspondem a

correções de arquivos ASCII em fita de papel, são convenientes os seguintes tipos de comando:

- Listagem de arquivos
- Duplicação de arquivos
- Substituição de registros
- Inserção de registros
- Eliminação de registros

b) Sendo frequente o trabalho de duplicação de fitas, tanto em formato ASCII como em formato binário, é desejável um comando de duplicação de arquivos.

c) Como um grande número de correções corresponde à modificação de uma sequência de caracteres dentro de um dado registro, são também desejáveis os seguintes tipos de comando:

- Substituição de sequências de caracteres em um registro
- Inserção de sequência de caracteres em um registro
- Eliminação de sequência de caracteres de um registro

d) Para facilitar a identificação de uma fita perfurada, é conveniente que se disponha de um comando de rotulação, que permita criar, no início de uma fita, um campo para identificação da mesma.

Note-se que todas estas decisões foram tomadas levando-se em conta uma configuração sem memória de massa. Em uma versão do editor que utilize, por exemplo, um disco, é muito conveniente que se disponha de vários recursos adicionais, tais como:

- Cópia de arquivos

- Comparação de arquivos
- Sequencialização de arquivos
- Ordenação dos registros de um arquivo ("sort")
- Inclusão, em um arquivo sequencializado, de tre chos também sequencializados de outros arquivos ("merge")
- Utilização do conteúdo de arquivos como programa de edição para outro(s) arquivo(s) ("macro")

É importante observar que, quando se dispõe de um disco, é comum que haja um sistema operacional presente. Embora não seja propriamente uma tarefa específica do editor, é conveniente que o sistema proporcione ao usuário algumas facilidades adicionais - tais como:

- Utilização do conteúdo de arquivos como sequên cia de comandos para o sistema operacional

- Utilização de procedimentos ("procedures"), sequên cias de comandos pré-programados, chamáveis por nome, com ou sem parâmetros. Para isso é necessário que haja uma grande integração entre os programas editor e sistema operacional desde a época do projeto até a fase final de implementação, uma vez que os "proce dimentos" nada mais são que macros que, depois de expandidos, trans formam-se em sequências de comandos para o sistema operacional. A expansão de tais macros é, no entanto, um trabalho típico do edi tor.

A versão do editor ora implantada foi projetada para fun cionar na configuração sem disco, e conta apenas com instruções de implementação relativamente simples. Mesmo assim, dispõe-se, nes ta versão, de comandos de listagem, duplicação, substituição, in serção e eliminação de registros, duplicação de fita binária, e rotulação de fitas.

6.3 - A Lógica do Editor

O programa implementado para a manipulação de registros, mos trado esquematicamente na fig. 6.3.1, é bastante simples, e consta basica

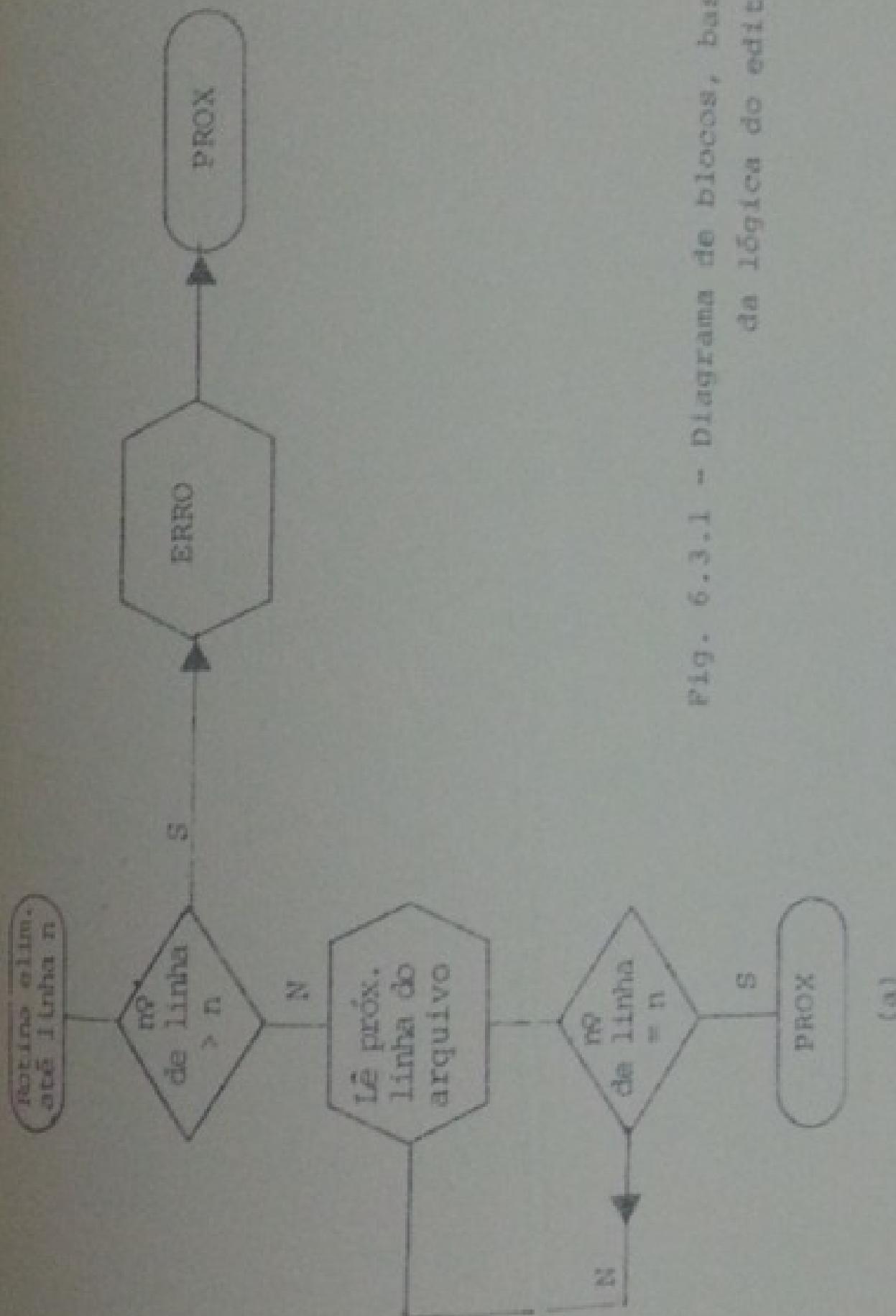



Fig. 6.3.1 - Diagrama de blocos, bastante simplificado, da lógica do editor simbólico

(a) Rotina de eliminação das linhas compreendidas entre a corrente e a *linha n*

(b)

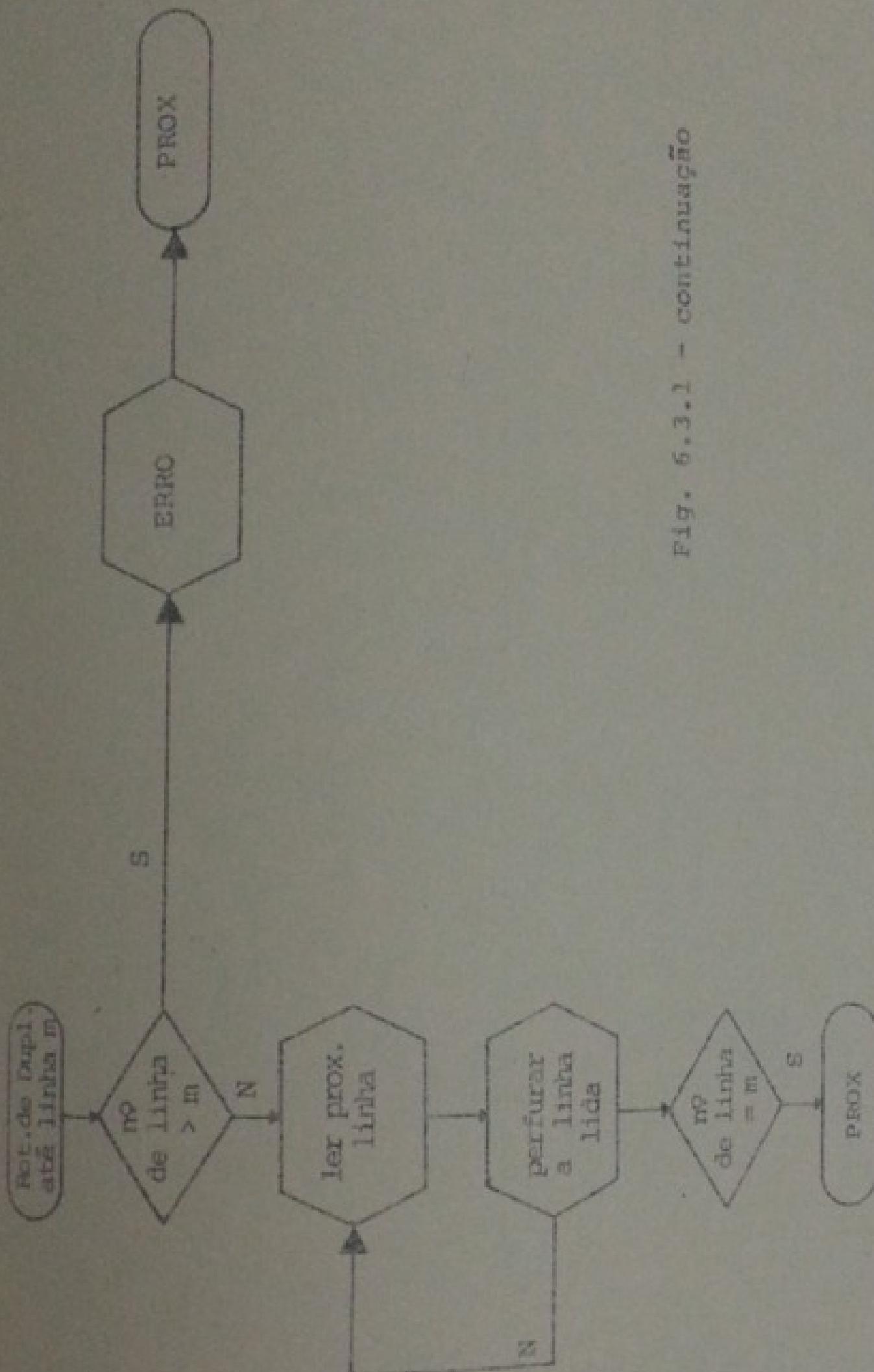



Fig. 6.3.1 - continuação

(b) Rotina de duplicação das linhas compreendidas entre a corrente e a *linha m*

EDITOR

Dar valores iniciais às variáveis do Programa nº de linha ← 1

PROX

Ler 1 Comando na Console

INTERPR.

É comando válido ?

N

ERRO

PROX

S

1 DUPLICAR FITA

2 ELIMINAÇÃO DAS LINHAS m a n

3 INSERÇÃO APÓS LINHA m

4 SUBSTITUIÇÃO DAS LINHAS m a n

5 DUPLICAR FITA BINÁRIA

NÚMERO DA LINHA

Imprimir o nº linha na console

PROX

ROTULAÇÃO

Gera um rótulo na fita saída

PROX

TÉRMINO DE EDIÇÃO

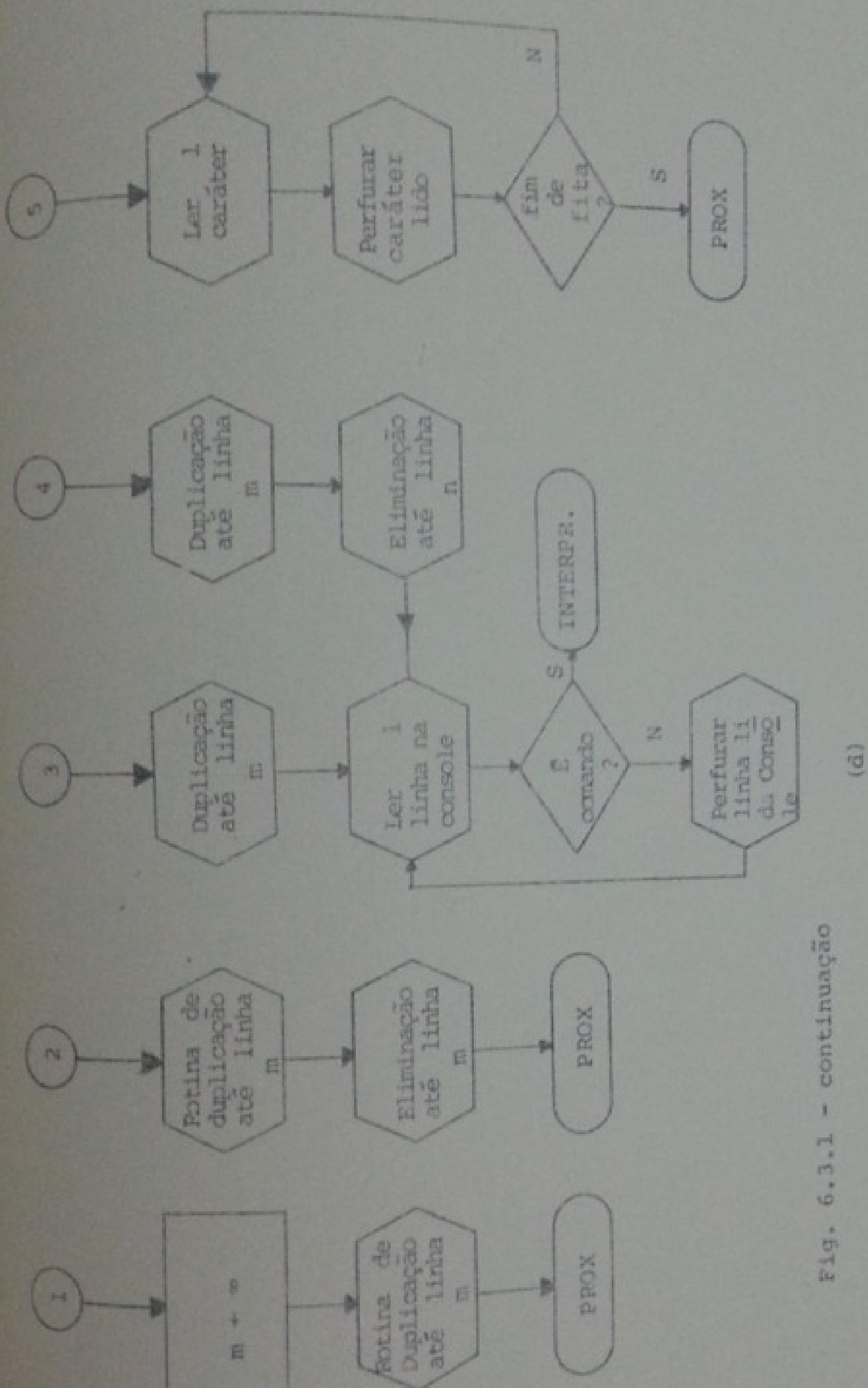
Pára o computador

(c)

Fig. 6.3.1 - continuação

(c) Programa principal. Estão mostrados apenas os quatro comandos mais importantes do editor.

Fig. 6.3.1 - continuação

(d)

(d) Programa principal. Estão mostrados apenas os quatro comandos mais importantes do editor.

mente de 2 rotinas, uma rotina de duplicação e uma rotina de eli minação de registros, as quais, executadas na sequência conveniente, com a ajuda de rotinas auxiliares, perfazem as funções das instruções escolhidas para alteração em nível de linha, conforme a descrição a seguir:

a) <u>Duplicação de Fita ASCII</u> - Para duplicar uma fita ASCII, basta que seja chamada sucessivamente a rotina de duplicação que lê um registro do arquivo fonte e gera um registro igual em um novo arquivo.

b) <u>Substituição de Registros</u> - Para substituir um registro, ou um conjunto de registros especificados, duplica-se o arquivo até o registro imediatamente anterior ao primeiro registro a substituir; elimina-se em seguida os registros subsequentes até o último desejado inserindo-se em seguida os registros fornecidos como dados para a substituição.

c) <u>Eliminação de Registros</u> - Para eliminar um registro ou conjunto de registros duplica-se o arquivo até o registro imediatamente anterior ao primeiro registro a eliminar; a seguir, chama-se a rotina de eliminação a qual se encarrega de ler apenas o conjunto especificado de registros, sem duplicá-los.

d) <u>Inserção de Registros</u> - Para se inserir novos registros após um registro especificado, duplica-se o arquivo até o registro desejado, e em seguida lê-se o conjunto de registros a inserir, duplicando-os.

e) <u>Listagem de Arquivos</u> - Para listar um arquivo, basta proceder analogamente à duplicação, chamando-se entretanto uma rotina de numeração dos registros imediatamente antes de sua duplicação.

f) <u>Duplicação de Fita Binária</u> - A instrução de reprodução de fita binária simplesmente duplica qualquer fita fornecida, fornecendo cópia idêntica à fita fornecida. Pode ser usada tanto para fitas fonte como para fitas objeto, e é implementada como uma simples sequência de leitura e escrita de caracteres não efetuando teste algum que não seja o fim-de-fita.

q) Rotulação - A instrução de rotulação parte de um texto lido na console e reproduz o mesmo na fita de papel, em formato legível. Isto é feito caráter a caráter, por meio de busca em tabelas que contêm o conjunto de caracteres necessário a rotulação, em formato interno codificado.

6.4 - Um exemplo de implementação

Descreve-se agora um exemplo de implementação de um editor simbólico para o Patinho Feio. Este programa foi escrito na linguagem de montador absoluto, e utiliza como unidade de entrada do arquivo origem uma leitora ótica. Utiliza como unidade de saída do arquivo destino uma "teletype", e como unidade de entrada das instruções e das alterações, uma segunda "teletype".

Antes de mais nada, deve-se definir os recursos de que se deseja dotar o editor simbólico para que este possa cumprir a finalidade a que se propõe. Começa-se por definir os formatos de entrada e da saída, e identificar os elementos a que se deseja ter acesso. Como o objetivo principal do editor é o da modificação de programas fonte, os registros de entrada serão sequências de caracteres ASCII terminados pela sequência de caracteres "return", "linefeed". Naturalmente, os registros de saída serão do mesmo tipo. Utilizando a mesma convenção usada no montador, "rubout" anula a linha que o contém. É sobre sequências de registros assim definidos que o editor vai trabalhar. Deve-se definir agora quais manipulações serão permitidas, afim de se estabelecer o conjunto de instruções do editor. Primeiramente, deseja-se poder modificar o conteúdo do arquivo, em nível de linha. Para isso, deve-se ter basicamente três comandos:

- Eliminação de Linhas
- Inserção de novas Linhas
- Substituição de Linhas

É frequente desejar-se eliminar um conjunto de registros sucessivos, ou então substituí-los por uma outra sequência de registros.

Assim, pode-se permitir que se especifiquem dois limites

para a eliminação ou para substituição. Estes limites são no
caso, identificados pelos números das linhas inicial e final a
eliminar ou substituir. Se for declarado apenas um número, subem
tende-se que se deseja eliminar ou substituir apenas o registro
cujo número foi especificado. No caso de inserção, deve ser for
necido apenas o número da linha depois da qual se deseja inserir
um ou mais registros.

Em segundo lugar, pode-se ter comandos para:

- Duplicação de arquivo
- Listagem de arquivo
- Fim de Edição

Antes da edição propriamente dita, pode-se utilizar o
comando de listagem para se obter uma numeração das linhas do tex
to a modificar, com o que fica facilitado o trabalho de escrever
os comandos da edição propriamente dita. O comando de fim de edi
ção pode ser utilizado simplesmente para duplicar uma fita, ou
então para finalizar uma sequência de comandos de edição, e sua
função é a de simplesmente copiar o restante do arquivo de entra
da no arquivo de saída.

Como o dispositivo de entrada dos comandos é o mesmo da
entrada das correções, e como o arquivo a modificar pode ser por
sua vez uma sequência de comandos para o editor, ou então um outro
texto no qual o primeiro carâter pode ser uma barra, torna-se con
veniente criar um meio pelo qual o editor não interprete o regis
tro de correção como sendo um comando. Para isto, como foi visto,
convencionou-se que, se aparecer uma sequência"//"nos dois primei
ros caracteres do registro de correção, este não será interpreta
do como comando, sendo a primeira barra ignorada.

Normalmente, a entrada das correções é manual, porém é
possível preparar uma fita de comandos e correções para o editor,
caso em que será utilizada a leitora de fitas do terminal como uni
dade de entrada. Neste caso, o final de linha deverá constar da
sequência de caracteres (_return_, _X-OFF_, _Linefeed_) ao invés de sim
plesmente _return_, _linefeed_, para que a leitura funcione corretamen
te. Dependendo do terminal utilizado, a sequência correta entre
return, _X-OFF_ e _Linefeed_ pode variar. Para evitar que o usuário
tenha que se preocupar com estes detalhes, foram criados dois co

mandos, os quais permitem duplicar uma fita com o programa de edição em formato aceito pelos terminais "Teletype" e "Olivetti" respectivamente. O que deve ser feito é apenas perfurar a fita normalmente, duplicando-a com o uso de um destes dois comandos, antes de utilizá-la para a edição definitiva. Observe-se que a geração desta fita pode ser feita "on line", bastando que se escolha a configuração conveniente para a operação do editor. Além disso, para facilitar a entrada manual das correções, introduziu-se a possibilidade de apagar os últimos caracteres da linha corrente.

Escrevendo-se uma sequência de caracteres especiais "ETX" e/ou "A controlado" serão apagados tantos caracteres do final da linha, quantos forem os "ETX" e/ou "A controlado" que constam da sequência.

Quando a correção é feita manualmente, sem a utilização de uma fita perfurada, às vezes é conveniente que se interrompa uma operação em andamento, ou que se reinicie o processamento. Para isto, utiliza-se o botão de interrupção do painel, que, quando pressionado, faz com que logo após a entrada ou saída corrente o processamento seja desviado para a parte do programa que pede comandos ao operador. Nesta ocasião, é interessante que o operador possa saber o número do próximo registro do arquivo origem a ser processado. Para tanto, criou-se um comando que pede ao editor o número de linha corrente. Como resultado, será impresso na console, o número da linha em questão.

Um outro comando serve para interromper a edição corrente e reiniciar a execução do programa editor.

Este comando não duplica o restante do arquivo de entrada. Para que isto seja executado, é necessário utilizar antes do comando citado um comando de finalização de edição. Todos os comandos de edição vistos até aqui operam com registros completos. Para o caso de programas fonte para o montador, entretanto, é convenientemente que se tenha a possibilidade de acrescentar ou substituir o campo de comentários das linhas de um programa já pronto. Para facilitar esta tarefa, criou-se um outro comando, com dois operandos, os quais indicam os limites dos registros cujo campo de comentários se deseja modificar. Os novos comentários devem ser fornecidos como se fossem novos registros, nos casos de inserção ou substituição e serão acrescentados às linhas convenientes, em substituição aos comentários existentes.

Obs: No caso de o primeiro caráter do comentário inserido ser "/", não é necessário que seja fornecido o comando "//", pois entre os limites fornecidos no comando de modificação dos comentários, o editor não interpreta a linha de edição como comando, mas apenas como texto a acrescentar à linha editada, no campo de comentários.

6.5 - Comentários sobre alguns detalhes de Implementação

O editor simbólico, sendo um programa que utiliza o sistema da interrupção do computador no controle da sua lógica, pode ter sua execução truncada durante a fase de edição. Este fato leva ao problema da paralização de um processo de entrada ou saída de um registro antes de sua finalização produzindo assim leitura ou perfuração incompleta de um registro. No caso da entrada do registro, se a interrupção ocorrer antes de a fita ser freiada pelo programa, o desvio para a rotina de tratamento de interrupção pode fazer com que o movimento de fita saia do controle do programa perdendo-se portanto, a posição do registro lido. No caso de saída de um registro, se a interrupção acontecer antes de o registro ser reproduzido completamente, o desvio para a rotina da interrupção e reinício do processo poderá fazer com que o próximo registro na fita seja prejudicado pela presença da parte não completamente reproduzida do registro anterior.

Para resolver estes problemas, lançou-se mão do próprio sistema de interrupção do computador, liberando-o e inibindo-o - convenientemente segundo as necessidades, por programa. Assim, adotou-se a política de inibir a interrupção do sistema antes da leitura de um registro, e liberá-la somente nos intervalos entre as saídas de dois registros. Com este procedimento, evitam-se os problemas apresentados, sincronizando-se o atendimento da interrupção com o início da saída de um registro.

Em termos de comunicação com o operador, o editor simbólico permite operação manual ou automática. Assim, dispondo-se da leitora mecânica de fita de papel de um terminal comum, pode-se - utilizar o editor em operação automática preparando previamente uma fita de papel com o programa de correção na linguagem do editor. O formato desta fita é semelhante ao formato da linha do programa de correção em operação manual, exceto que o final de linha

Na tabela 6.5.1 estão listados todos os comandos ora disponíveis no editor implementado.

Embora projetado para a configuração mínima com dois terminais e uma leitora de fita perfurada, o editor tem sido utilizado numa configuração mais versátil, com leitora de fita, terminal, perfuradora rápida e impressora de linha, o que torna muito mais eficiente a sua operação.

Para isto, assim como para qualquer adaptação a outras configurações, foi introduzida uma secção de configuração, cuja função é a de ler, no painel de chaves, os números das interfaces de entrada e saída correspondentes aos diversos dispositivos de perfuração, entrada de comandos, etc, e, com estas informações, modificar as rotinas correspondentes de entrada e saída do editor.

Com algumas adaptações, é possível tornar o editor parte de um sistema operacional. Neste caso, as rotinas de entrada e saída poderão trabalhar sobre arquivos em disco ao invés de arquivos em fita de papel. A utilidade de um programa como este torna-se muito grande se houver uma concordância suficiente entre os formatos dos dados por ele manipulados e os dos dados aceitos pelos outros componentes do sistema, uma vez que o editor pode ser, se devidamente bem projetado, o programa mais utilizado do sistema operacional (ref. 13).

APÊNDICE 1. O CONJUNTO DAS INSTRUÇÕES DE MÁQUINA DO PATINHO FEIO

APÊNDICE 1 - O CONJUNTO DAS INSTRUÇÕES DE MÁQUINA DO PATINHO FEIO

Com a finalidade de facilitar os algoritmos de montagem, as instruções de máquina foram divididas em seis grupos, que são detalhadas a seguir. A notação usada para os códigos neste texto, será a hexadecimal, a menos de ressalva em contrário.

A1.1 - Grupo 1 - Instruções de Referência À Memória

A característica principal das instruções de referência à memória é a de ocupar duas palavras e apresentar dois campos, um de quatro bits, que representa o código de operação, e outro de doze bits, que representa um endereço de memória.

A disposição destes bits é a da fig. A1.1, onde está representada esquemàticamente uma instrução de CARREGA--ACUMULADOR com campo de endereço de memória igual a 2_{10}.

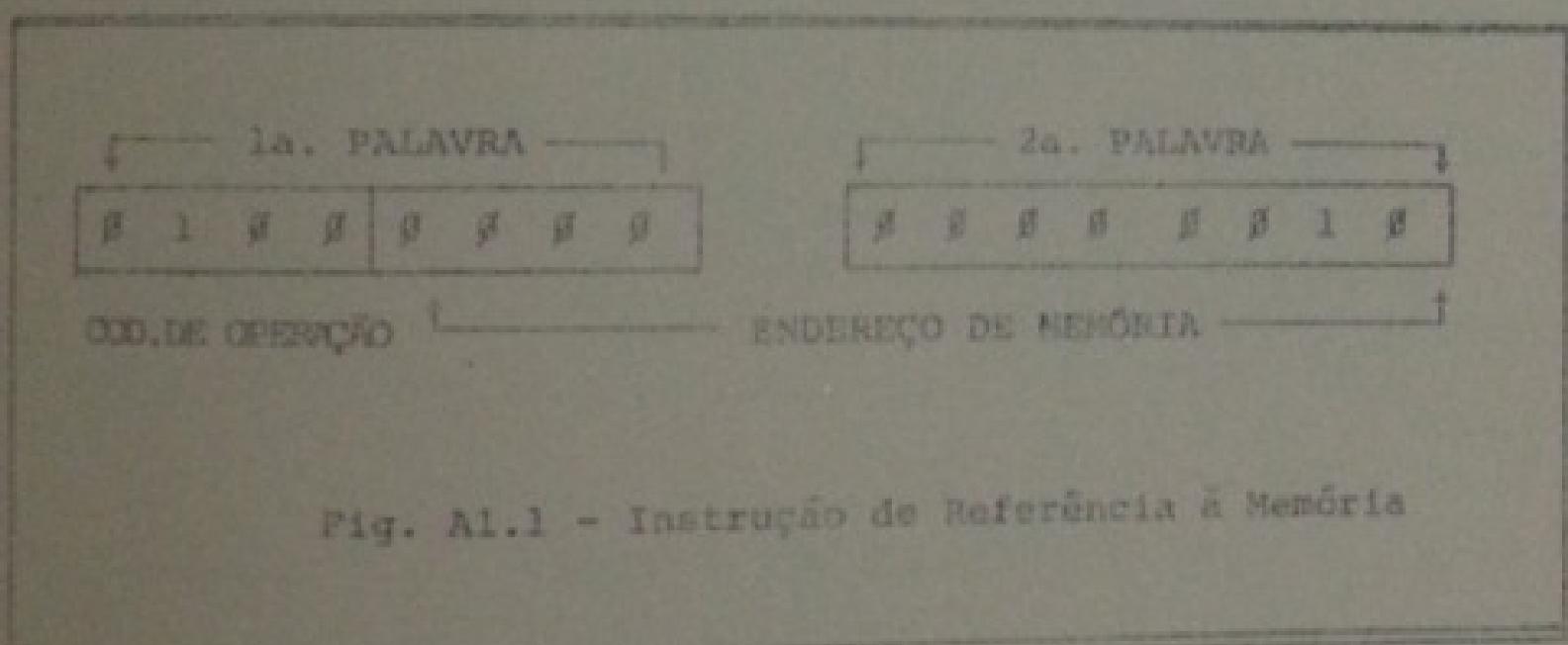



Fig. A1.1 - Instrução de Referência à Memória

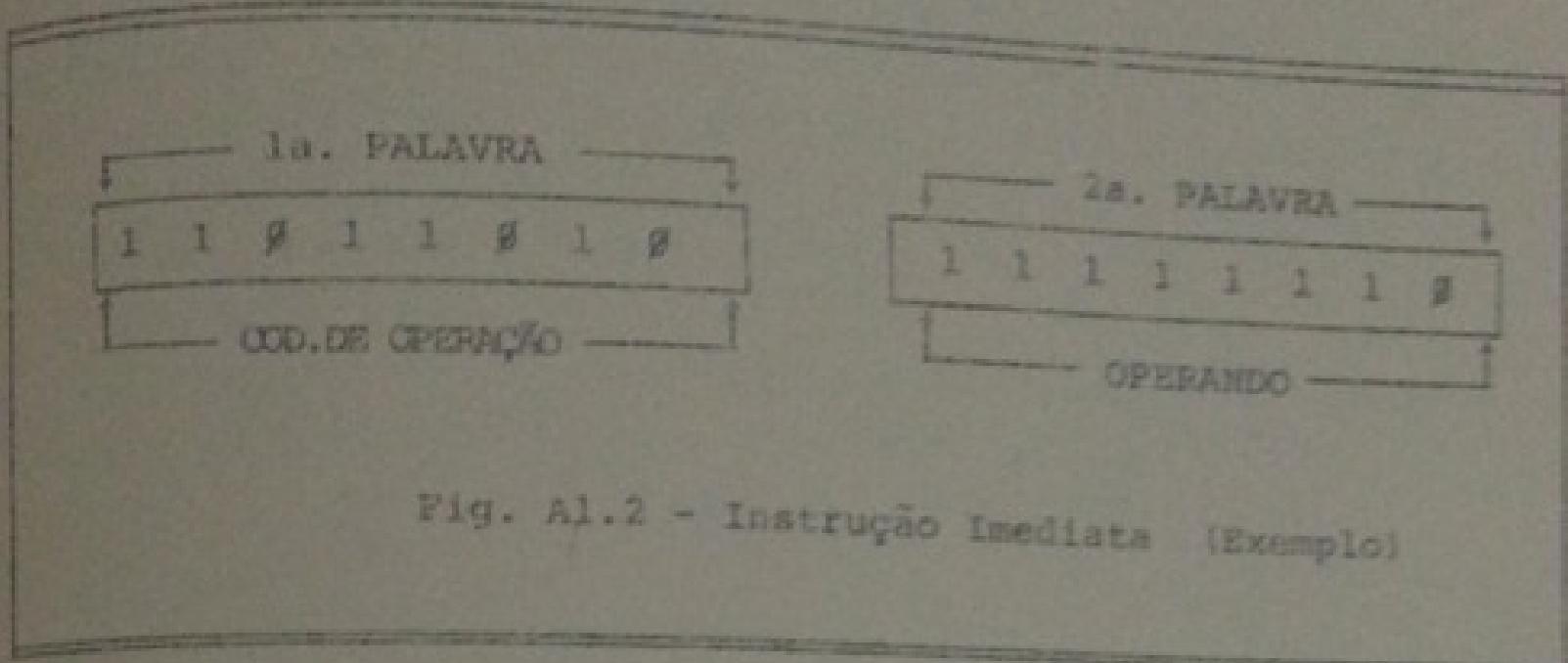
As Instruções de referência à memória e seus correspondentes mnemônicos e códigos de operação em hexadecimal estão apresentados na tabela A1.1.

CÓDIGO DE OPERAÇÃO	MNEMÔNICO	INSTRUÇÃO
Ø	PLA	Pulo Incondicional
1	PLAX	Pulo Indexado
2	ARM	Armazena Acumulador
3	ARMX	Armazena Indexado
4	CAR	Carrega Acumulador
5	CARX	Carrega Indexado
6	SOM	Soma
7	SOMX	Soma Indexada
A	PLAN	Pula Se acumulador é Negativo
B	PLAZ	Pula Se acumulador é Zero
E	SUS	Subtrai Um ou Salta
F	PUG	Pula e Guarda

Tabela A1.1 - Instrução de Referência à Memória

A montagem do código objeto de uma instrução de referência à memória consiste em, calculado o endereço referenciado, em 12 bits, concatená-lo com o código de operação correspondente à instrução em questão, o qual é fornecido pela rotina de geração da parte invariante do código objeto.

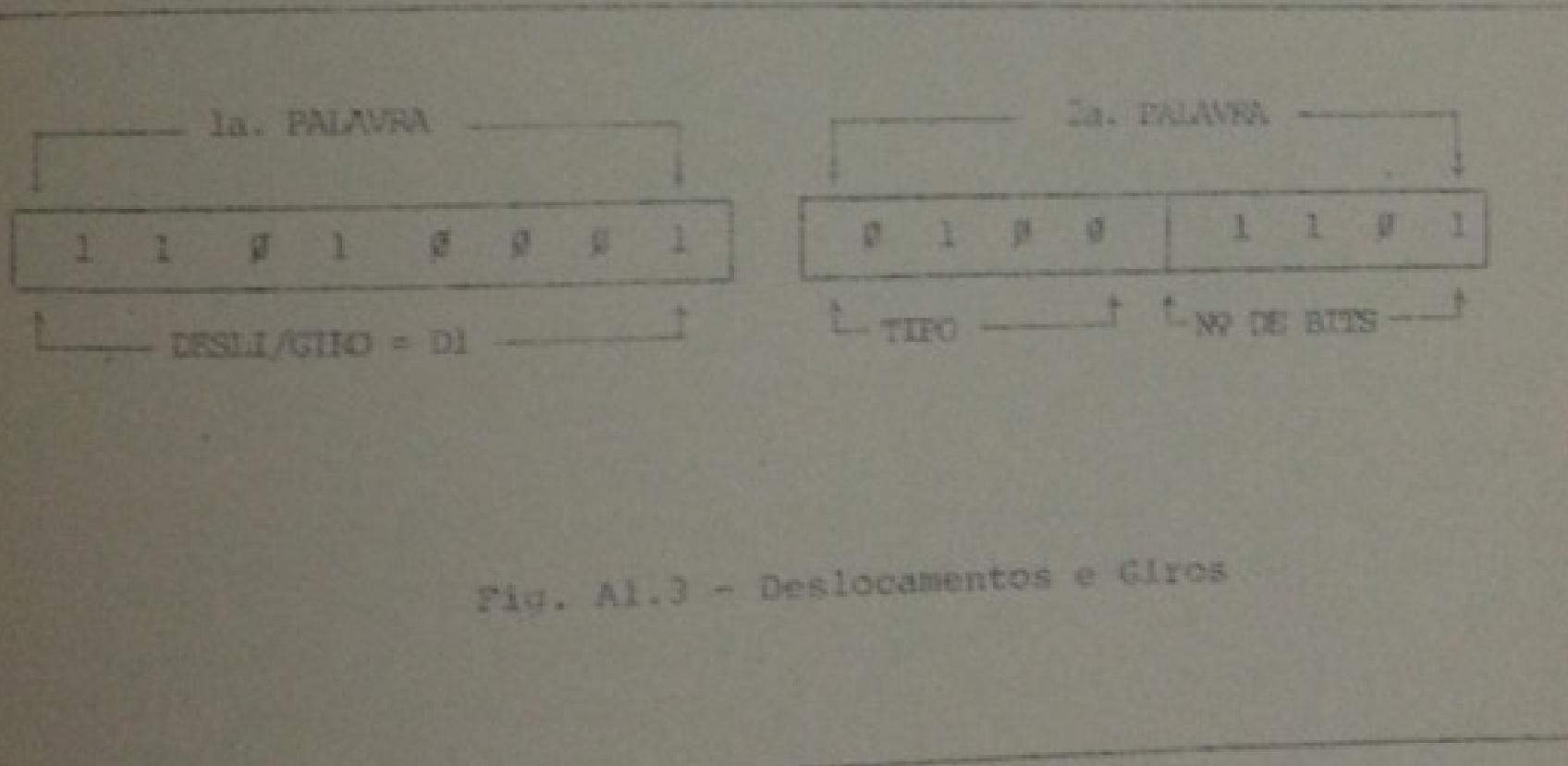
A1.2 - Grupo 2 - Instruções de Endereçamento Imediato

As instruções de endereçamento imediato também ocupam duas palavras, mas são formadas de dois campos de oito bits cada um. O primeiro campo é um código de operação estendido, e indica qual a instrução, e o segundo campo é o operando imediato. Na fig. A1.2, está representada uma instração de CARREGA-IMEDIATO, com operando igual a -2_{10}.

Fig. A1.2 - Instrução Imediata (Exemplo)

Na tabela A1.2 estão listadas as quatro instruções imediatas disponíveis no Patinho Feio.

CÓDIGO DE OPERAÇÃO	MNEMÔNICO	INSTRUÇÃO
D2	XOR	Ou Exclusivo Imediato
D4	NAND	"NAND" Imediato
D8	SOMI	Soma Imediata
DA	CARI	Carrega Imediato

Tabela A1.2 - Instruções Imediatas

O algoritmo de montagem deste grupo consiste em justapor ao código de operação correspondente à instrução o operando imediato especificado, após a devida conversão para número binário de 8 bits.

A1.3 - Grupo 3 - Instruções de Deslocamentos e Giros

As instruções de deslocamento e giros ocupam duas palavras, sendo que a primeira contém sempre o código D1, que identifica o grupo.

A segunda palavra é formada de dois campos de 4 bits cada um, sendo que os primeiros 4 bits representam o tipo de deslocamento ou giro, e os 4 últimos indicam de quantos bits deve ser o deslocamento ou giro, sendo que, neste campo, cada bit "1" representa o deslocamento ou giro de um bit.

A fig. A1.3, representa uma das possíveis implementações de um deslocamento para a esquerda de 3 bits.

Fig. A1.3 - Deslocamentos e Giros

TIPO	MNEMÔNICO	INSTRUÇÃO
Ø	DD	Deslocamento à Direita
1	DDV	Deslocamento à Direita com V
2	GD	Giro à Direita
3	GDV	Giro à Direita com V
4	DE	Deslocamento à Esquerda
5	DEV	Deslocamento à Esquerda com V
6	GE	Giro à Esquerda
7	GEV	Giro à Esquerda com V
8	DDS	Deslocamento à Direita (com duplicação de sinal)

Tabela A1.3 - Deslocamentos e Giros

Algoritmo de Montagem: a primeira palavra é sempre D1. A segunda é formada pelos 4 bits de tipo, de acôrdo com a tabela A1.3, seguidos de 4 bits construídos de modo tal que o número de bits "1" seja igual ao número de deslocamentos ou giros desejado.

A1.4 - Grupo 4 - Instruções de Entrada e Saída

As instruções de entrada e saída são também formadas de duas palavras, contendo quatro campos, conforme a fig. A1.4.

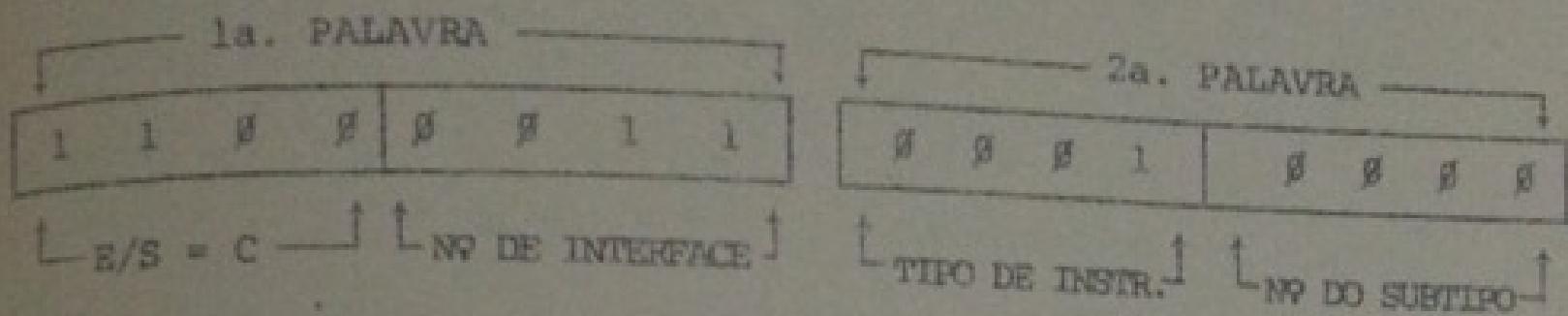



Fig. A1.4 - Entrada e Saída

• O primeiro campo identifica o grupo de entrada e saída, e contém sempre o código "C".

• O segundo campo especifica o número da interface à qual se refere o comando de entrada e saída. No exemplo, trata-se da interface número 3. Este campo é especificado na linguagem do montador, como os 4 primeiros bits do operando da instrução.

• O terceiro campo identifica qual o tipo de instrução de entrada e saída a executar. Este campo está implícito no mnemônico utilizado para representar a instrução, na linguagem do montador. No exemplo da fig. A1.4 este campo vale 1, e portanto, trata-se de uma instrução de FUNÇÃO, segundo a tabela A1.4.

• O quarto campo serve para identificar, dentro de um mesmo tipo de instruções de entrada e saída, qual a instrução específica a ser executada. Na linguagem do montador, este campo é especificado pelos 4 últimos bits do operando da instrução.No exemplo,este campo vale Ø. Na linguagem do montador, esta instrução seria codificada como FNC/3Ø.

TIPO DE INSTRUÇÃO	MNEMÔNICO	INSTRUÇÃO
1	FNC	Função
2	SAL	Salto Condicional
4	ENTR	Entrada de Dado
8	SAI	Saída de Dado

Tabela A1.4 - Entrada e Saída

O algoritmo de montagem para as instruções deste grupo consiste em acrescentar às informações recebidas da rotina de geração da parte invariante do código, (que no caso são o 1º e o 3º campos da instrução), as informações provenientes do operando, o qual deve ser desmembrado em dois dados de 4 bits e anexado aos outros dois campos para formar a instrução completa.

A1.5 - Grupo 5 - Instruções Curtas Com Operando

As instruções deste grupo ocupam uma única palavra, e podem ser divididas em dois subgrupos:

- as instruções de leitura do painel
- as instruções de saltos condicionais segundo e estado dos "flipflops" T e V.

As instruções de painel tem o formato mostrado na fig. A1.5.1, e constam de 3 campos.

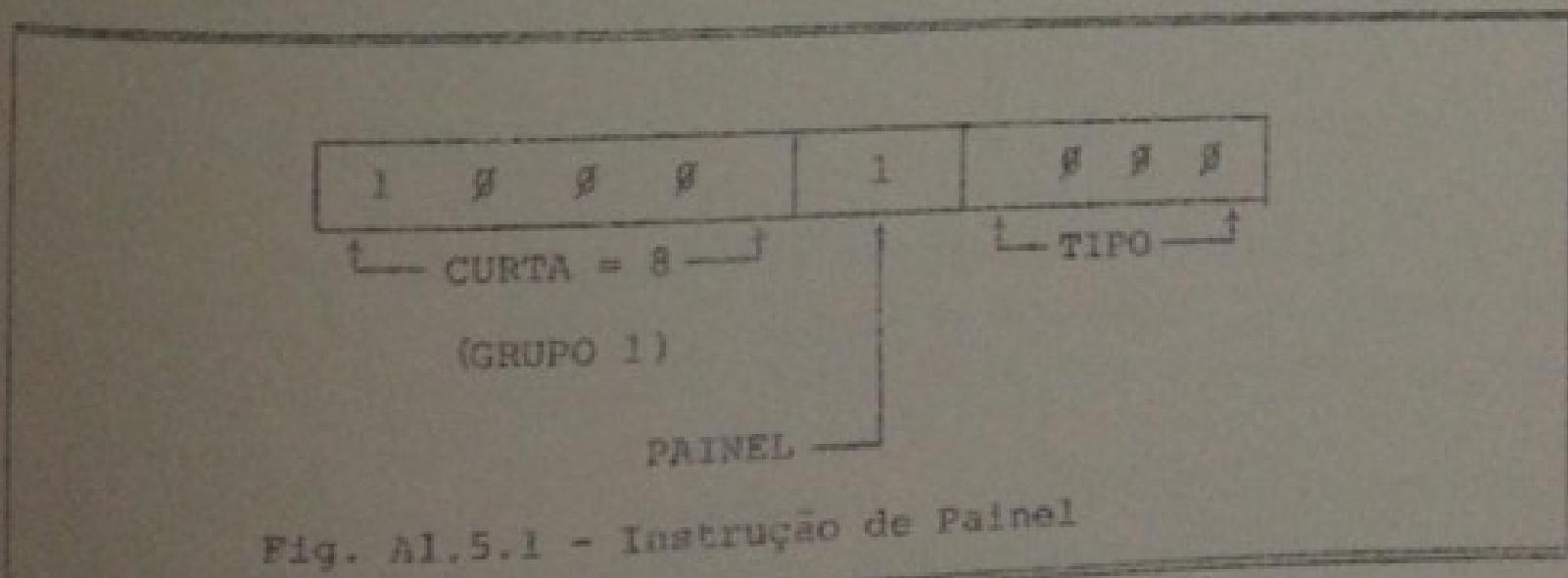
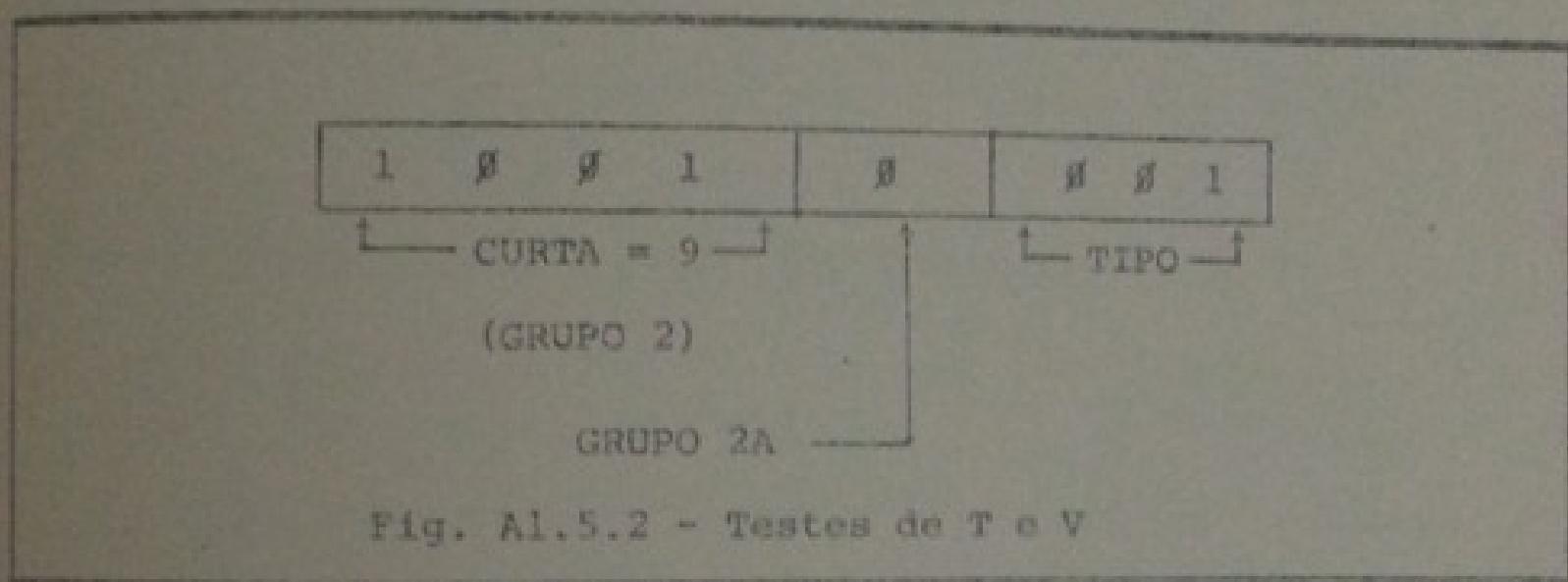



Fig. A1.5.1 - Instrução de Painel

- O 1º campo, formado de 4 bits, indica que a instrução é curta do grupo 1 e contém sempre o código 8

- O 2º campo indica que se trata de instrução de Painel, e é formada de um bit apenas, igual a "1".

- O 3º campo é formado de 3 bits, e indica o tipo de instrução de painel a executar, e contém o número especificado como operando da instrução, em 3 bits. No exemplo, trata-se de uma instrução PNL Ø.

As instruções de saltos condicionais segundo o estado de T e V formam o grupo 2A e seu formato está representado na fig. Al.5.2, e constam de 3 campos.

Fig. Al.5.2 - Testes de T e V

- O primeiro, de 4 bits, identifica a instrução como curta do grupo 2, e contém sempre o código 9.

- O 2º campo é formado de um único bit, e identifica o grupo A, e é sempre "Ø".

- O 3º campo especifica o tipo de teste a executar.

A montagem destas instruções consiste em desligar ou ligar o bit menos significativo da palavra fornecida pela rotina de geração da parte invariante do código, conforme o operando da instrução seja ou não igual a zero.

Na tabela Al.5 tem-se a listagem de todas as instruções curtas que exigem operando, com os respectivos códigos em hexadecimal.

CÓDIGO	MNEMÔNICO	OPERANDO	INSTRUÇÃO
88 a 8F	PNL	Ø a 7	Painel
9Ø	SV	Ø	Salta se V = Ø
91	SV	1	Salta se V = 1
92	SVM	Ø	Salta e Muda se V = Ø
93	SVM	1	Salta e Muda se V = 1
94	ST	Ø	Salta se T = Ø
95	ST	1	Salta se T = 1
96	STM	Ø	Salta e Muda se T = Ø
97	STM	1	Salta e Muda se T = 1

Tabela A1.5 - Instruções Curtas Com Operando

A1.6 - Grupo 6 - Instruções Curtas Sem Operando

A este grupo pertencem as restantes instruções curtas do grupo 1, bem como as do grupo 2B.

As instruções curtas do grupo 1 obedecem ao formato mostrado na fig. A1.6.1, com três campos, dois dos quais são constantes, com 4 e 1 bits, e valem 8 e Ø , respectivamente.

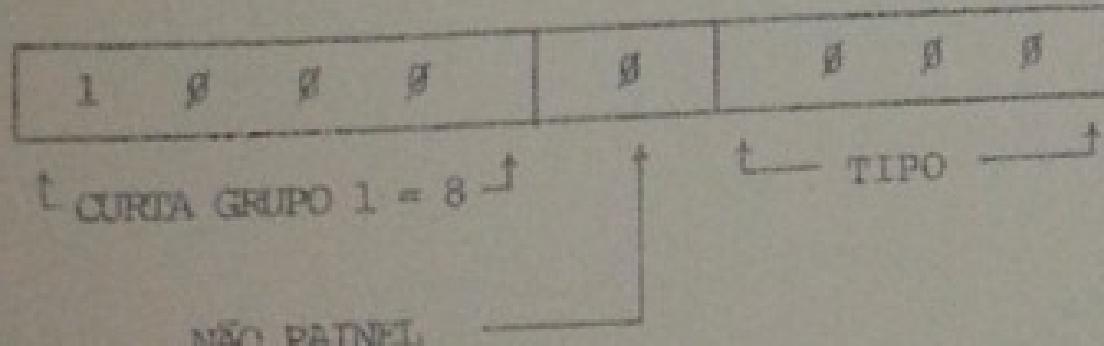



Fig. A1.6.1 - Instrução Curta, Grupo 1

O terceiro campo tem 3 bits, e especifica o tipo da instrução, conforme está mostrado na tabela A1.6.1.

TIPO	MNEMÔNICO	INSTRUÇÃO
Ø	LIMPØ	Limpe Acumulador e V
1	UM	Faz AC = 1 e V = Ø
2	CMP1	Complemento de 1
3	CMP2	Complemento de 2
4	LIM	Limpa V
5	INC	Incrementa AC
6	UNEG	Faz AC = -1 e V = Ø
7	LIMP1	Limpa Acum. e faz V = 1

Tabela A1.6.1 - Instruções Curtas do Grupo 1 S/Operando

As instruções curtas do grupo 2B obedecem ao formato mostrado na fig. A1.6.2.

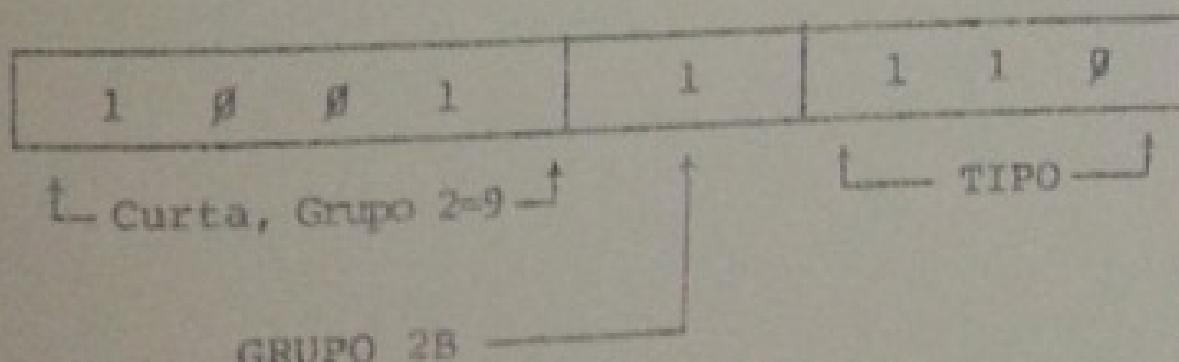



Fig. A1.6.2 - Instrução Curta, Grupo 2B

Possui também 3 campos, sendo os dois primeiros constantes com 4 e 1 bits, e valendo 9 e 1 respectivamente. O campo de TIPO, com 3 bits, identifica a instrução de acordo com a tabela Al.6.2.

TIPO	MNEMÔNICO	INSTRUÇÃO
ø	PUL	Pula e Limpa Interrupção
1	TRE	Troca AC com EXT
2	INIB	Inibe Interrupção
3	PERM	Permite Interrupção
4	PARE	Pare
5	ESP	Espera
6	TRI	Troca Acumulador c/Índice
7	IND	Indireto

Tabela Al.6.2 - Instruções Curtas Grupo 2B

A montagem das instruções do grupo 6 é feita por pesquisa direta em tabela, pois há correspondência direta entre o código gerado e o mnemônico usado.

APÊNDICE 2. ROTINAS AUXILIARES UTILIZADAS NA CONSTRUÇÃO DO "SOFTWARE" BÁSICO

APÊNDICE 2 - ROTINAS AUXILIARES UTILIZADAS NA CONSTRUÇÃO DO "SOFTWARE" BÁSICO

Estão descritos neste apêndice as rotinas que foram utilizadas na fase de elaboração dos primeiros programas do "software" básico. Estas rotinas foram escritas inicialmente em linguagem de máquina, e tiveram o objetivo de facilitar a utilização da máquina para o desenvolvimento de outros programas.

A2.1 - A Rotina de Geração de Fita Perfurada Carregável ("dumper")

Enquanto um computador ainda não tem a possibilidade de gerar uma fita perfurada a partir de um programa escrito em uma linguagem de que disponha, a única maneira de guardar os programas já escritos é por meio de suas listagens (fonte e binária). Cada vez que for necessário carregar estes programas, esta operação é executada manualmente pelas chaves do painel, perdendo-se o programa toda vez que se modificar o conteúdo das posições de memória por ele ocupadas, tornando-se necessário portanto recarregá-lo, pelas chaves, a cada utilização.

Para atenuar este problema, é interessante que se disponha de um meio simples de salvar o conteúdo da memória em formato carregável por programa. Com esta finalidade construiu-se um programa gerador de fita de papel à imagem da memória, cuja finalidade é a de produzir, numa fita de papel, uma cópia do conteúdo da memória, além de algumas outras informações simples cujo objetivo é permitir que o conteúdo da fita seja carregado corretamente quando da utilização da mesma para recarga do programa.

Dados o endereço inicial e o comprimento da informação a salvar em fita, a rotina divide o conteúdo da memória em blocos, cada um dos quais tem a estrutura descrita no apêndice A3 (formato da fita absoluta).

A2.2 - O Carregador de Fitas Absolutas

Geradas as fitas com o conteúdo de memória, por meio do "dumper", é necessário um programa que consiga recarregar correta

mente a memória a partir do seu conteúdo. Para isto foi desenvolvido um carregador de fitas absolutas que aceita os dados no formato produzido pelo gerador de fitas à imagem da memória e com eles restaura o conteúdo das posições especificadas de memória. Devido à sua grande utilização, este programa permanece residente na memória,na área protegida, para que possa ser utilizado a qualquer momento . Sendo relativamente extenso (128 palavras), seu carregamento manual é trabalhoso, sendo útil para carregá-lo um outro programa - mais curto, mas de uso específico, que deve ser executado toda vez que se desejar guardar na memória protegida o carregador de fitas absolutas,a partir de uma fita perfurada.

Este pré-carregador ocupa 35 posições de memória sendo portanto muito mais simples seu carregamento pelo painel que o do carregador propriamente dito.

Maiores detalhes sobre carregador encontram-se na ref.2

A2.3 - A Rotina de Listagem do Conteúdo da Memória

Quando se trabalha em linguagem de máquina, alterando instruções e dados de memória constantemente através do painel, é importante que se possa obter, a qualquer momento, uma listagem do conteúdo da memória para verificação. Com esta finalidade desenvolveu-se uma rotina que, a partir dos endereços externos de memória cujo conteúdo se deseja conhecer, forneça uma listagem em hexadecimal de tais conteúdos. Esta é basicamente composta de uma rotina de conversão binário-hexadecimal, que vai lendo sequencialmente as posições de memória especificadas e escrevendo a seguir seu conteúdo, em hexadecimal, na console.

A2.4 - A Rotina de Carregamento de Memória a partir de dados em Hexadecimal

De grande utilidade na fase de construção do "software" básico em linguagem de máquina é a rotina de carregamento da memória a partir de dados, escritos na console, em hexadecimal. Sua finalidade é a de substituir o carregador de fita binária quando não se tem ainda construída tal fita. Escrito o programa, e devidamente convertido para linguagem de máquina, em notação hexadecimal, pode-se utilizar esta rotina para carregá-lo na memória a

partir de um endereço que também é fornecido à rotina. Outra utilidade para tal programa é a de modificar posições de memória de um programa em fase de depuração.

Para isso, deve-se fornecer o endereço da posição a modificar, bem como o novo conteúdo desta posição e das seguintes, repetido o processo para cada correção a fazer.

Uma vez depurado o programa, pode-se utilizar o gerador de fita perfurada à imagem de memória para obter uma fita, em formato carregável, do programa em questão. Esta foi a técnica utilizada para obter as primeiras cópias dos programas desenvolvidos em linguagem de máquina no Patinho Feio, enquanto o montador não era operacional.

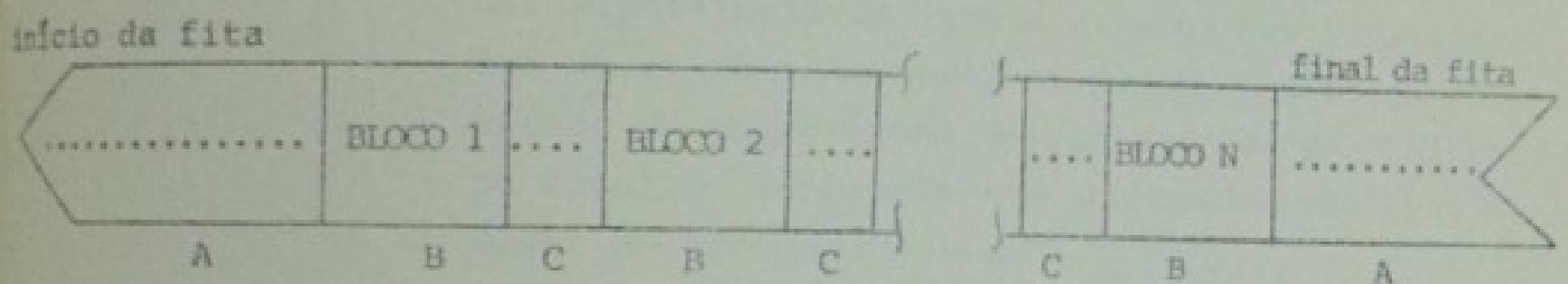
A2.5 - Comentários

- Na fase de depuração de programas, sejam estes escritos em linguagem de máquina ou na linguagem do montador, programas do tipo dos descritos neste apêndice são muito importantes para facilitar a tarefa do programador. Entretanto, tais programas obviamente ocupam memória, portanto quando o programa a depurar é muito grande logo se apresenta o problema de falta de espaço, na memória, para a coexistência dos dois programas. Normalmente, programas grandes ocupam áreas grandes para dados, tabelas, e outras informações,importantes apenas durante a execução dos mesmos. Contorna-se em parte o problema de falta de espaço na memória carregando os programas de utilidade ou programas de teste em tais áreas de dados, com o que o programa que está sendo depurado permanece intacto, podendo-se portanto copiá-lo numa fita, tirar listagens em hexadecimal, modificar posições de memória e mesmo testar rotinas novas.

Constata-se assim ser conveniente a existência de várias versões dos programas de utilidade, cada qual funcionando em uma região de memória, para não restringir a liberdade de organização da memória ao programador.

- Dispondo-se de um outro computador é óbvio que a maioria das rotinas descritas possam ser substituídas por programas que, funcionando neste outro computador, venham a automatizar tais procedimentos. Por exemplo, a existência de um montador para a lin

guagem do Patinho Feio, escrita num outro computador, dispensa a programação em linguagem de máquina bem como a rotina de geração de fita à imagem de memória. A existência de um interpretador em outra máquina de maior capacidade coloca também à disposição do programador recursos adicionais muito importantes para a depuração de programas, como "traces", "dumps" de memória, etc, sem a utilização da própria memória do Patinho Feio para comportar os respectivos programas de utilidade (cap. 2,3).

Além disso, a utilização de outro computador pode por à disposição do programador todos os recursos de que tal computador dispõe, como armazenamento secundário e periféricos rápidos, o que é muito interessante no caso.

- É conveniente que uma rotina básica de utilidade usada com tanta frequência como é o carregador binário, devido à sua característica de programa residente, seja preservada o máximo possível. Isto é obtido, em parte, mantendo-a na região protegida de memória. Entretanto, uma proteção total seria obtida construindo-se uma memória do tipo ler-somente com o carregador binário, o que dispensaria totalmente o uso do painel para carregá-lo, automatizando completamente a operação de carregamento de programas. Uma vantagem adicional seria a da liberação, para possível utilização por programas normais, das 128 posições de memória atualmente ocupadas pelo carregador binário.

APÊNDICE 3. FORMATOS DE FITA OBJETO

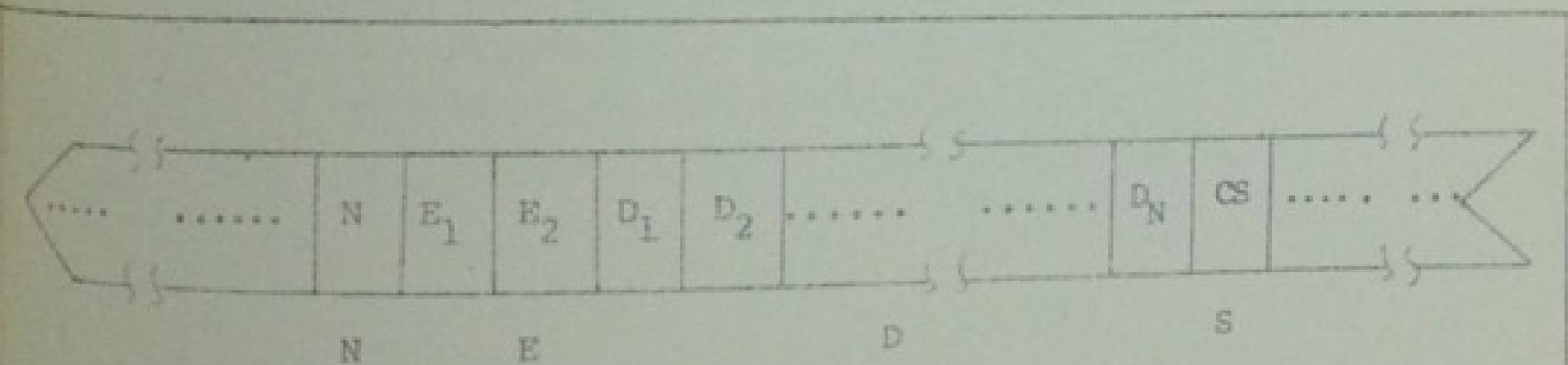
APÊNDICE 3 - FORMATOS DE FITA OBJETO

A3.1 - Formato da fita Objeto Absoluta

O formato da fita objeto absoluta está vinculado ao programa carregador absoluto (ref. 2), e deverá obedecer à estrutura mostrada na fig. A3.1.1, onde se nota uma sequência de blocos de dados, separados entre si por sequências de quatro nulos, e precedida e sucedida por trilhas de"bytes"nulos (início e final da fita).

A - Trilha de"bytes"nulos (no mínimo 2Ø)
B - Blocos de Dados (v.fig. A3.1.2)
C - Separação entre Blocos (4"bytes"nulos)

Fig. A3.1.1 - Estrutura de uma fita Objeto Absoluta

Coda bloco de dados deverá apresentar a estrutura mostrada na fig. A3.1.2, onde se notam quatro regiões:

N - formada de 1"byte"não nulo, indica o número de "bytes"de região D.

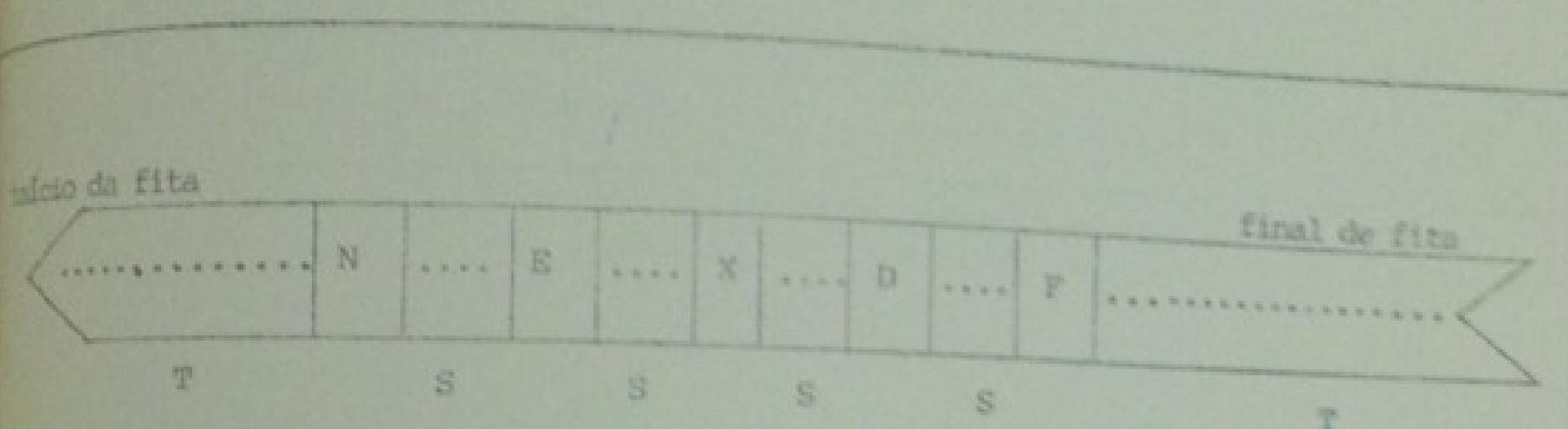
E - formada de 2"bytes", contém o endereço de memória onde deverá ser carregado o primeiro dos "bytes"de região D. Este endereço é formado pelos doze bits menos significativos da sequên

cia E1, E2. Os quatro"bytes"mais significativos são nulos.

D - esta região é a que contém a informação a ser carregada na memória. Seu comprimento é variá vel, e especificado na região N. Os dados D_1, D_2 ... D_N serão armazenados pelo carregador absoluto a partir do endereço especificado na região E.

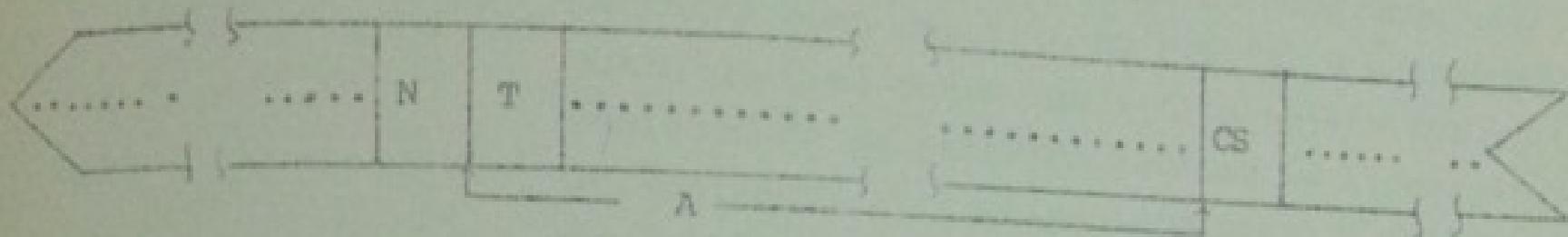
S - esta região é formada por um único"byte",construí do de tal maneira que a soma de todos os "bytes" do bloco, incluindo CS, tenha seus oito bits me nos significativos iguais a zero. Sua finalidade é a de evitar, ao menos em parte, o carregamento errado de um programa devido a erros de leitura na fita, e é normalmente conhecido como"byte"de teste de soma ("checksum").

N - Número de"bytes"de região D

E - Endereço inicial de armazenamento dos dados

D - Sequência de dados

S -"Byte"de teste de soma ("checksum")

Fig. A3.1.2 - Estrutura do bloco de dados absoluto

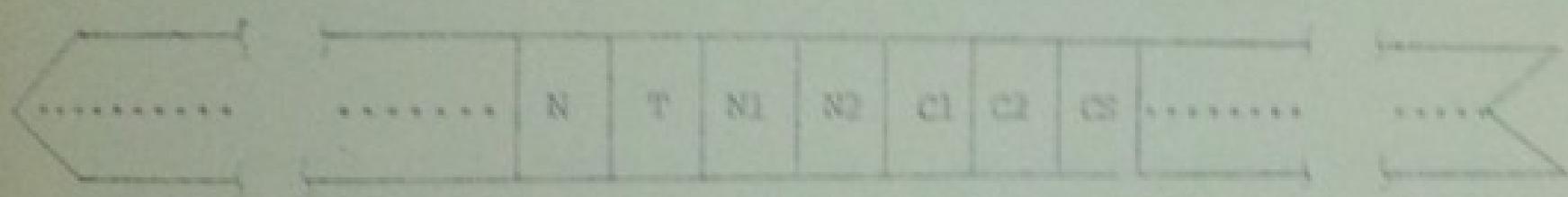
A3.2 - Formato da fita Objeto Relocável

Uma fita objeto relocável (ref. 9) produzida pelo montador relocável ou por um compilador deverá ter a estrutura mostrada na fig. A3.2.1.

S - Separação entre blocos (4 bytes nulos)
N - Bloco de NOME (único, obrigatório)
E - Sequência de blocos ENT (opcional)
X - Sequência de blocos EXT (opcional)
D - Sequência de blocos de dados (opcional)
F - Bloco de FIM (único, obrigatório)
T - Trilhas de "bytes" nulos (20 no mínimo)

Fig. A3.2.1 - Estrutura de uma fita objeto relocável

Cada bloco, independentemente de sua função, deverá ter a estrutura apresentada na fig. A3.2.2.

N - Número de"bytes"contidos na região A
T - Tipo de bloco (v. tabela A3.2.1)
CS - "Byte"de teste de Soma ("checksum")

Fig. A3.2.2 - Formato geral de um bloco relocável

Em cada bloco, o"byte"T (fig. A3.2.2) indica o tipo de bloco, segundo a tabela A3.2.1.

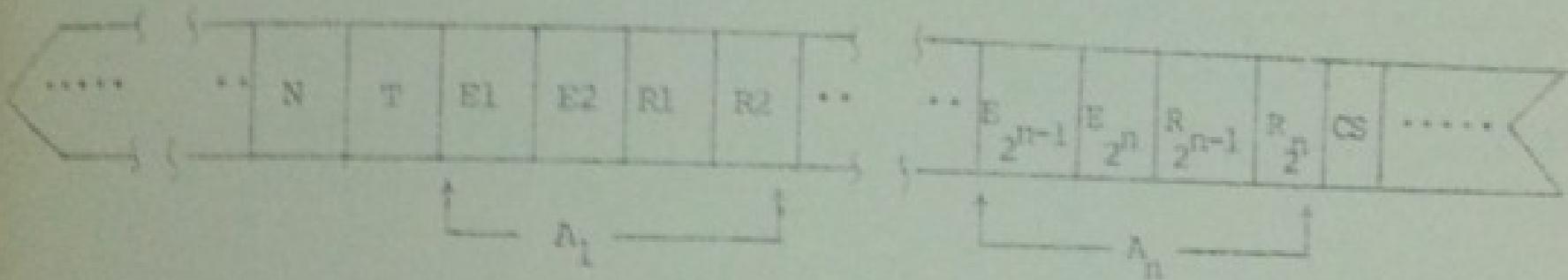
VALOR DE T	TIPO DO BLOCO	OBSERVAÇÕES
∅	NOME	Programa Principal
2	NOME	Segmento
4	NOME	Subrotina
6	ENT	
8	EXT	
1∅	DADOS	
12	FIM	

Tabela A3.2.1 - Significado do Campo T

Descreve-se a seguir o formato de cada um dos cinco tipos de bloco relocável.

a) <u>Bloco de NOME</u>: Neste bloco (fig. A3.2.3) apresentam-se duas informações básicas: o nome do programa, compactadas como descrito no cap. 2, e o comprimento de área comum. Se a fita objeto for proveniente de um programa escrito na linguagem do montador, a primeira informação é colhida das pseudos NOME, SEGM ou SUBR, e a segunda, das pseudos COM.

N = 5

T = Ø, 2 ou 4, conforme se trate de programa principal, segmento ou subrotina

N1, N2 - Nome compactado do programa

C1, C2 - Comprimento de área comum

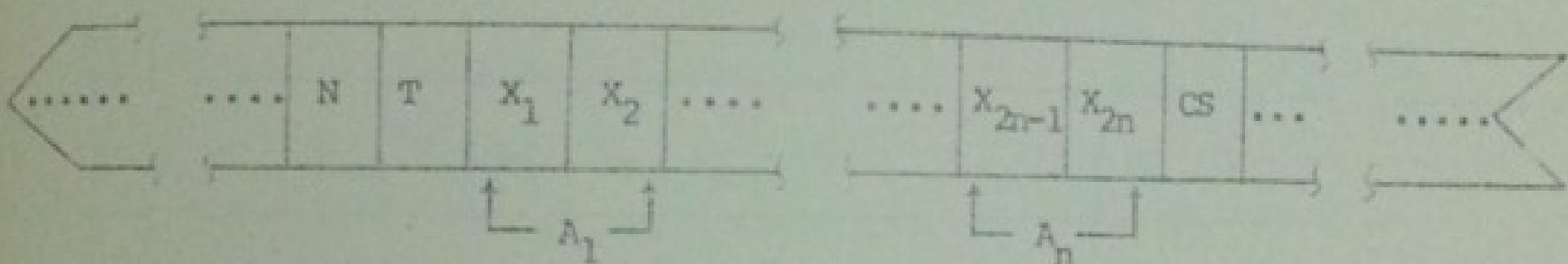
CS - "Byte"de soma ("checksum")

Fig. A3.2.3 - Bloco de NOME

b) <u>Bloco de ENT</u>: Neste bloco (fig A3.2.4) são declarados os nomes dos pontos de entrada (nomes globais) contidos no programa. Para fitas objeto provenientes de programas escritos na linguagem do montador, tais nomes são colhidos a partir dos operandos das pseudos ENT. Cada bloco pode conter mais de um nome de ponto de entrada.

N = 4n + 1, onde n é o número de nomes globais declarados no bloco

T = 6

$A_1, \ldots, A_n$ - são os grupos de "bytes" que definem os n nomes globais

E_{2i-1} , E_{2i} - nome do i-ésimo ponto de entrada global compactado

R_{2i-1} , R_{2i} - endereço relocável do i-ésimo ponto de entrada

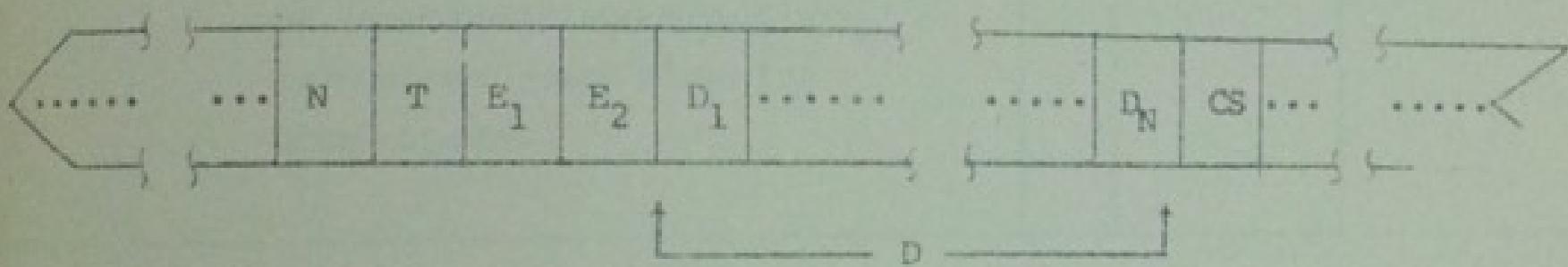
CS - "Byte" de teste de soma ("checksum")

Fig. A3.2.4 - Bloco de ENT

c) <u>Bloco de EXT</u>: Neste bloco (fig. A3.2.5) são declarados os nomes globais externos referenciados no programa. No caso de a fita objeto ter sido produzida pelo montador, estas informações terão sido extraídas a partir dos operandos das pseudos EXT. Cada bloco poderá conter mais de um nome global externo.

N = 2n + 1, onde n é o número de nomes globais declarados no bloco

T = 8

A_1, ... An - são os n nomes globais declarados

X_{2i-1} , X_{2i} - i-ésimo nome global, compactado

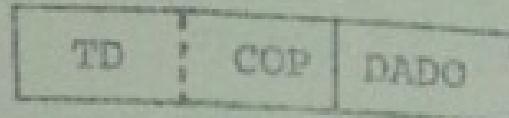
CS - "Byte"de teste de soma ("checksum")

Fig. A3.2.5 - Bloco de EXT

d) Bloco de Dados: Um bloco de dados tem o formato mos-trado na fig. A3.2.6.

Os dados (campo D) estão detalhados na fig. A3.2.7.

N = número total de"bytes"do bloco, menos dois

T = 1Ø

E_1, E_2 - Endereço relocável do primeiro dado da sequência D

D - Sequência de N dados D_1 , ... D_N (fig. A3.2.7)

CS - "Byte"de teste de soma ("checksum")

Fig. A3.2.6 - Formato do Bloco de Dados

, dado absoluto: "2 bytes"

TD	COP	DADO

, dado não absoluto: "3 bytes"

TD	COP	OPERANDO

Fig. A3.2.7 - Formatos dos dados:

Os operandos relocáveis e apresentam-se como endereços relativos de 12 bits.

Os operandos externos apresentam-se como símbolos compactados.

TD	COP	Observações
1	ø	Dado absoluto
ø	código de operação	Operando relocável
2	código de operação	Operando comum
4	código de operação	Operando externo (compactado)

e) Bloco de FIM: O bloco de fim (fig. A3.2.8) contém como informação o endereço de execução do programa principal ou segmento.

No caso de subrotina, este campo é preenchido com zeros.

Fig. A3.2.8 - Bloco de FIM

O bloco de FIM é produzido quando for encontrada uma pseudo FIM em programas escritos na linguagem do montador. O endereço de execução é extraído do operando deste pseudo, caso não se trate de subrotina.

Para maiores detalhes sobre o uso das fitas objeto, consultar as refs. 2 e 1Ø.

APÊNDICE 4. ALGUNS EXEMPLOS DE UTILIZAÇÃO DOS PROGRAMAS DESENVOLVIDOS

```
#PLT
 ORG /6
INIC CARI 10 ACERTA
 ARM CONT CONTADOR
 LIMPO * ACERTA
 TRI * INDICE
LOOP CARX LINHA PEGA CARACTER
 PUG SAI IMPRIME
 TRI * APONTA
 INC * PROXIMO
 TRI * CARACTER
 SUS CONT VERIFICA SE ACABOU
 PLA LOOP NAO : CONTINUA IMPRIMINDO
 PARE * SIM : PARA
 PLA INIC SE PARTIDA : VOLTA IMPRIMIR
*
CONT DEFC 0 CONTADOR
LINHA DEFC #P MENSAGEM A SAIR
 DEFC #A
 DEFC #I
 DEFC #O
 DEFC #     *
 DEFC #F
 DEFC #E
 DEFC #I
 DEFC #O
 DEFC /0D  CR
 DEFC /0A  LF
*
SAI PLA 0
 SAI /A0 SAI DADO
 SAL /A1 ESPERA TERMINO DE SAIDA
 PLA *-2
 PLA SAI RETORNA AO PROGRAMA PRINCIPAL
*
 FIM INIC
```

A4-1 - Listagem de um Programa Fonte Absoluto

```
INC  006
COT  01A
LOP  00C
LIA  01B
SAI  026

/00 SI

PASSO2

  1  @BLT
  2  006                ORG     /6
  3  006 0A 0A   INIC   CARI    10       ACERTA
  4  008 20 1A          ARM     CONT     CONTADOR
  5  00A 80             LIMPO   *        ACERTA
  6  00B 9E             TRI     *        INDICE
  7  00C 50 1B   LOOP   CARX    LINHA    PEGA CARACTER
  8  00E F0 26          PUG     SAI      IMPRIME
  9  010 9E             TRI     *        APONTA
 10  011 85             INC     *        PROXIMO
 11  012 9E             TRI     *        CARACTER
 12  013 E0 1A          SUS     CONT     VERIFICA SE ACABOU
 13  015 00 0C          PLA     LOOP     NAO : CONTINUA IMPRIMINDO
 14  017 90             PARE    *        SIM : PARA
 15  018 00 06          PLA     INIC     SE PARTIDA : VOLTA IMPRIMIR
 16  *
 17  01A 00      CONT   DEFC    0        CONTADOR
 18  01B 50      LINHA  DEFC    @P       MENSAGEM A SAIR
 19  01C 41             DEFC    @A
 20  01D 54             DEFC    @T
 21  01E 4F             DEFC    @O
 22  01F 20             DEFC    @
 23  020 46             DEFC    @F
 24  021 45             DEFC    @E
 25  022 49             DEFC    @I
 26  023 4F             DEFC    @O
 27  024 0D             DEFC    /0D      CR
 28  025 0A             DEFC    /0A      LF
 29  *
 30  026 00 00   SAI    PLA     0        SAI DADO
 31  028 CA 80          SAI     /A0      ESPERA TERMINO DE SAIDA
 32  02A CA 21          SAL     /A1
 33  02C 00 2A          PLA     *-2      RETORNA AO PROGRAMA PRINCIPA
 34  02E 00 26          PLA     SAI
 35  *
 36  006                FIM     INIC
```

A4-2 - Saída do Montador Absoluto usando como original a fita A4-1

```
 ORG /006
AAG CARI /0A
 ARM ABK
 LIMPO
 TRI
AAM CARX ABL
 PUG ACG
 TRI
 INC
 TRI
 SUS ABK
 PLA AAM
 PARE
 PLA AAG
ABK PLA AFA
 CAR BFE
 CAR PCA
 CAR GEF
 CAR JEP
 PLA NAK
ACG PLA AAA
 SAI /A0
ACK SAL /A1
 PLA ACK
 PLA ACC
ABL EQU /01A
AFA EQU /050
BFE EQU /154
PCA EQU /F20
GEF EQU /645
JEP EQU /94F
NAK EQU /00A
AAA EQU /000
 FIM 0
```

A4-3 - Saída do Desmontador Absoluto a fita original é a que foi obtida em A4-2

```
LIST.?

PERF.?

FITA?
TABELA DE SIMBOLOS
ARK   /01A
ABL   /01B
ACG   /026
AAM   /00C
AAG   /006
AFA   /050
BFE   /154
PCA   /F20
GEF   /645
JEP   /94F
NAK   /00A
AAA   /000
ACK   /02A
            FITA?


                             ORG       /006
006  CA 0A  AAG              CARI      /0A          010
008  20 1A                   ARM       ARK
00A  80                      LIMPO     *
00B  9E                      TRI       *
00C  50 1B  AAM              CARX      ABL
00E  F0 26                   PUG       ACG
010  9E                      TRI       *
011  85                      INC       *
012  9E                      TRI       *
013  E0 1A                   SUS       ARK
015  00 0C                   PLA       AAM
017  9D                      PARE      *
018  00 06                   PLA       AAG
01A  00 50  ARK              PLA       AFA
01C  41 54                   CAR       BFE
01E  4F 20                   CAR       PCA
020  46 45                   CAR       GEF
022  49 4F                   CAR       JEP
024  00 0A                   PLA       NAK
026  00 00  ACG              PLA       AAA
028  CA 80                   SAI       /80
02A  CA 21  ACK              SAL       /A1
02C  00 2A                   PLA       ACK
02E  00 26                   PLA       ACG
            ABL              EQU       /01B
            AFA              EQU       /050
            BFE              EQU       /154
            PCA              EQU       /F20
            GEF              EQU       /645
            JEP              EQU       /94F
            NAK              EQU       /00A
            AAA              EQU       /000
                             FIM       0
```

A4-4 - Saída do Desmontador Absoluto usando como original a fita obtida em A4-3.

```
/P
/R,2
 NOVE PATO
 EXT SAI
/I,29
 FIM INIC
/P
/P
/I,29
PRLT
 SUPR ESCR
 ENT SAI
*
/R,36
 FIM
/T
```

A4-5 - Programa de Edição para
a fita A4-1

```
/P
 ?/R,2
 ? NOME PATO
 ? EXT SAI
 ?/I,29
 ? FIM INIC
 ?/P
 ?/P
 ?/I,29
 ???LT
 ? SUBR ESCR
 ? ENT SAI
 ?*
 ?/R,36
 ? FIM
 ?/T
```

A4-6 - Listagem de Console do Editor simbólico uti
lizando como arquivo de Edição a fita A4-5

```
#L[illegible]
 NOME [illegible]
 ENT SAI
INIC [illegible] SE ACERTA
 ARM CONT CONTADOR
L[illegible] * ACERTA
 [illegible] * INICIO
LOOP CARR LINHA PEGA CARACTER
 [illegible] SAI IMPRIME
 [illegible] * APONTA
 [illegible] * PROXIMO
 [illegible] * CARACTER
 [illegible] CONT VERIFICA SE ACABOU
 PLA LOOP NAO : CONTINUA IMPRIMINDO
 PARE * SIM : PARA
 PLA INIC SE PARTIDA : VOLTA IMPRIMIR
*
CONT DEFC 0 CONTADOR
LINHA DEFC #P MENSAGEM A SAIR
 DEFC #R
 DEFC #T
 DEFC #C
 DEFC #     *
 DEFC #F
 DEFC #E
 DEFC #I
 DEFC #O
 DEFC /0D  CR
 DEFC /0A  LF
*
FIM INIC
```

A4-7 - Listagem da primeira parte da saída do editor simbólico. O arquivo editado é a fita A4-2 e o programa de edição é o da fita A4-5

```
#LI
 SUBR ESDA
 ENT SAI
*
SAI PLA 0
 SAI /A0 SAI DADO
 SAL /A1 ESPERA TERMINO DE SAIDA
 PLA *-2
 PLA SAI RETORNA AO PROGRAMA PRINCIPAL
*
FIM
```

A4-8 - Listagem de segunda parte da saída do editor simbólico. O arquivo estidado é o da fita A4-2 e o programa de edição é o da fita A4-5

```
PAO  ***  NOE
SAI  ***  EXT
INC  000
COT  014
LOP  006
LIA  015


/00 SI

PASSO2

 1                     RALT
 2  000                           NOME     PATO
 3  000                           EXT      SAI
 4  000 DA 0A          INIC       CARI     10        ACERTA
 5  002 20 14 R                   ARM      CONT      CONTADOR
 6  004 60                        LIMPO    *         ACERTA
 7  005 9E                        TRI      *         INDICE
 8  006 50 15 R        LOOP       CARX     LINHA     PEGA CARACTER
 9  008 F0 00 X                   PUG      SAI       IMPRIME
10  00A 9E                        TRI      *         APONTA
11  00B 65                        INC      *         PROXIMO
12  00C 9E                        TRI      *         CARACTER
13  00D E0 14 R                   SUS      CONT      VERIFICA SE ACABOU
14  00F 00 06 R                   PLA      LOOP      NAO : CONTINUA IMPRIMINDO
15  011 9D                        PARE     *         SIM : PARA
16  012 00 00 R                   PLA      INIC      SE PARTIDA : VOLTA IMPRIMIR
17                     *
18  014 00             CONT       DEFC     0         CONTADOR
19  015 50             LINHA      DEFC     @P        MENSAGEM A SAIR
20  016 41                        DEFC     @A
21  017 54                        DEFC     @T
22  018 4F                        DEFC     @O
23  019 20                        DEFC     @
24  01A 46                        DEFC     @F
25  01B 45                        DEFC     @E
26  01C 49                        DEFC     @I
27  01D 4F                        DEFC     @O
28  01E 0D                        DEFC     /0D       CR
29  01F 0A                        DEFC     /0A       LF
30                     *
31  000                           FIM      INIC
```

A4-9 - Saída do Montador Relocável usando como original a fita A4-7

```
ESR  ***  SUR
SAI  000  ENT


/00 SI

 PASSO2

   1                  RRLT
   2  000                        SUBR      ESCR
   3  000                        ENT       SAI
   4                  *
   5  000 00 00       SAI        PLA       0
   6  002 CA 60                  SAI       /A0       SAI DADO
   7  004 CA 21                  SAL       /A1       ESPERA TERMINO DE SAIDA
   8  006 00 04 R                PLA       *-2
   9  008 00 00 R                PLA       SAI       RETORNA AO PROGRAMA PRINCIPAL
  10                  *
  11  000                        FIM
```

A4-10 - Saída do Montador relocável usando como original a fita A4-8

```
TAB SIMB

     PAO   NOME
/000 ZZ[   COM
     SAI   EXT
/006 GAA
/014 EBA
/015 FBA


                                @BLT
0000                                  NOME   PAO
0001                            ZZ[   COM    /000
0002                                  EXT    SAI
0003  000  DADA                 SAI   CARI   /0A
0004  002  2014   R                   ARM    EBA
0005  004  80                         LIMPO
0006  005  9E                         TRI
0007  006  5015   R             GAA   CARX   FBA
0008  008  F000   X                   PUE    SAI
0009  00A  9E                         TRI
0010  00B  85                         INC
0011  00C  9E                         TRI
0012  00D  E014   R                   SUS    EBA
0013  00F  0006   R                   PLA    GAA
0014  011  9D                         PARE
0015  012  0000   R                   PLA    SAI
0016  014  00                   EBA   DEFC   /00
0017  015  50                   FBA   DEFC   /50
0018  016  41                         DEFC   /41
0019  017  54                         DEFC   /54
0020  018  4F                         DEFC   /4F
0021  019  20                         DEFC   /20
0022  01A  46                         DEFC   /46
0023  01B  45                         DEFC   /45
0024  01C  49                         DEFC   /49
0025  01D  4F                         DEFC   /4F
0026  01E  0D                         DEFC   /0D
0027  01F  0A                         DEFC   /0A
0028  000                             FIM    SAI
```

A4-11 - Saída do Desmontador relocável usando como entrada a fita obtida em A4-9

```
TAB SIMB

     ESR   SUBR
/000 ZZ[   COM
/000 SAI   ENT
/004 EAA


                                @BLT
0000                                  SUBR   ESR
0001                            ZZ[   COM    /000
0002                                  ENT    SAI
0003                                         /000
0004  000  0000                 SAI   PLA    /A0
0005  002  CA60                       FNC    /A1
0006  004  CA21                 EAA   FNC    EAA
0007  006  0004   R                   PLA    SAI
0008  008  0000   R                   PLA    SAI
0009  000                             FIM
```

A4-12 - Saída do Desmontador relocável usando como entrada a fita obtida em A4-10

```
      C    D     V     A   E   I   T   V   R    D
    ( 006  DADA  CAI   00  00  00  00  00
    | 008  201A  ARN   0A  00  00  00  00  01A  00  50
(1) | 00A  80    LIP   0A  00  00  00  00
    | 00B  9E    TRI   00  00  00  00  00
    ( 00C  501B  CAX   00  00  00  00  00  01B  50  41

(2) { B  00C  501B  CAX   00  00  00  00  00  01B  50  41

(3) { 00E  F026  PUG   50  00  00  00  00  026  00  00

(4) { B  02E  0026  PLA   50  00  00  00  00  026  00  10

      C    D     V     A   E   I   T   V   R    D
    ( 010  9E    TRI   50  00  00  00  00
    | 011  85    INC   00  00  50  00  00
(5) | 012  9E    TRI   01  00  50  00  00
    | 013  E01A  SUS   50  00  01  00  00  01A  0A  50
    | 015  000C  PLA   50  00  01  00  00  00C  50  1B
    ( 00C  501B  CAX   50  00  01  00  00  01C  41  54

(6) { B  00C  501B  CAX   50  00  01  00  00  01C  41  54

    ( 00E  F026  PUG   41  00  01  00  00  026  00  10
    | 026  CA80  SAI   41  00  01  00  00
(7) | 02A  CA21  SAL   41  00  01  00  00
    | 02C  002A  PLA   41  00  01  00  00  02A  CA  21
    ( 02E  0026  PLA   41  00  01  00  00  026  00  10

(8) { B  02E  0026  PLA   41  00  01  00  00  026  00  10

    ( 026  0010  PLA   41  00  01  00  00  010  9E  85
(9) | 010  9E    TRI   41  00  01  00  00
    ( 011  85    INC   01  00  41  00  00

         0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
     ( 00  00 00 08 0B 07 00 DA 0A 20 1A 80 9E 50 1B F0 26
     | 01  9E 85 9E E0 1A 00 0C 90 00 06 09 50 41 54 4F 20
     | 02  46 45 49 4F 0D 0A 00 10 CA 80 CA 21 00 2A 00 26
     | 03  47 22 6A 22 6B 22 64 90 8B 04 0F 82 B0 46 0B C0
     | 04  22 5C 22 5A 22 56 90 8B C4 00 62 B0 55 0B C0 22
     | 05  78 22 76 22 72 90 9A CE 1B 86 26 66 26 5F 26 5E
     | 06  26 60 26 61 80 26 5A 26 5C 81 26 5B 26 5D 9A DA
(10) | 07  3F F2 62 F2 02 86 26 66 46 7F DE D1 B0 AE 80 25
     | 08  C7 80 26 66 45 C7 D1 41 65 9E 55 B0 26 65 F0 00
     | 09  55 B0 26 64 9F 46 64 9E 81 99 80 F2 04 0B 50 20
     | 0A  A4 99 20 A5 56 64 F2 62 E0 00 00 A4 00 6F DA DA
     | 0B  9E 56 4D 66 80 80 C0 F0 00 00 B1 81 25 C7 00 81
     | 0C  9E D1 41 9E 10 C6 00 EC 01 5F 05 B3 00 E8 05 08
     | 0D  03 B5 03 CD 03 D9 01 1B 00 DE F1 51 01 2B F1 AD
     | 0E  F3 FB F2 A1 F5 7E 00 DE F1 51 00 6F DA 28 9E 9A
     ( 0F  CE 16 CE 21 00 F2 CE 40 B1 03 9E DA 28 9E F2 70

(11) { 012  9E    TRI   02  00  41  00  00
     { 013  E01A  SUS   41  00  02  00  00  01A  09  50
```

A4-13 - Saídas da Rotina de Depuração. O programa utilizado é o que foi obtido em A4-10

(1),(3),(5),(7),(9),(11) - "trace"
(2),(4),(6),(8)- mensagem de ponto de teste ("breakpoint")
(10) - "dump" de memória

(1)

```
:PR,PATO3
 ?
S
 ?
N
 ?
L
 ?
E,4
 ?
P
 ?
C
 CANAIS A DISPOSICAO,OPCAO:0-N. EM USO,3,8
 ITTY  IPERF  IIMP

R,0,3
 ?
E,FE0
 ?
L
 ?
V
 ?
P
 ?
C,000,050
 ?
E,10
 ?
X,10
 009E
 0085
 009E
 00E0
 001A
 0000
 000C
 009D
 0000
 0006
 0000
 0050
 0041
 0054
 004F
 0020
 ?
E,28
 ?
A,CR,
 ?
X,1
 0080
 ?
A,CR,
 ?
X,1
 0021
 ?
T,000,040
 ?
E,6
 ?
L
 ?
P
PATO FEIO
 ?
```

A4-14 - Saída do simulador-interpretador. O programa que está sendo simulado é o que foi obtido em A4-2

(1) - Listagem da Consolo

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	00	00	60	56	DA	1C	DA	0A	20	1A	80	9E	50	18	F0	26
01	9F	B5	9E	F0	1A	00	0C	9D	00	06	00	50	41	54	4F	20
02	46	45	49	4F	0D	0A	00	00	C8	80	C8	21	00	2A	00	26
03	9A	9D	76	9C	E1	D1	56	D0	2D	2D	57	21	2C	27	60	49
04	50	4A	2D	08	60	53	E9	CE	99	D2	6E	E8	67	01	76	98

A4-14 - Saída do simulador-interpretador. O programa que está sendo simulado é o que foi obtido em A4-2

(2) - "Dump" de memória

ENDE	INST	ACC	IND	EXT	T	V	EN.EF	CDANT	CDATU
0006	DA0A	0A	00	00	00	00	0A0A	5648	
0008	201A	0A	00	00	00	00	001A	0050	0A50
000A	80	00	00	00	00	00			
002C	002A	41	01	00	00	00	002A	CB21	
002A	CB21	41	01	00	00	00			
002C	002A	41	01	00	00	00	002A	CB21	
002A	CB21	41	01	00	00	00			
002C	002A	41	01	00	00	00	002A	CB21	
002A	CB21	41	01	00	00	00			
002C	002A	41	01	00	00	00	002A	CB21	
002A	CB21	54	02	00	00	00			
002E	0026	54	02	00	00	00	0026	0010	
0026	0010	54	02	00	00	00	0010	9EB5	
0010	9E	02	54	00	00	00			
0011	B5	03	54	00	00	00			
002A	CB21	20	04	00	00	00			
002E	0026	20	04	00	00	00	0026	0010	
0026	0010	20	04	00	00	00	0010	9E65	
0010	9E	04	20	00	00	00			
0011	E5	05	20	00	00	00			
0012	9E	20	05	00	00	00			
0013	E01A	20	05	00	00	00	001A	0650	0550
002A	CB21	4F	06	00	00	00			
002C	002A	4F	08	00	00	00	002A	CB21	
002A	CB21	4F	08	00	00	00			
002C	002A	4F	08	00	00	00	002A	CB21	
002A	CB21	00	09	00	00	00			
002C	002A	0C	09	00	00	00	002A	CB21	

(3)

A4-14 - Saída do simulador-interpretador. O programa que está sendo simulado é o que foi obtido em A4-2

(3) - Trecho de "trace"

R E F E R Ê N C I A S

1. FREGNI, EDSON
"Projeto Lógico da Unidade de Controle de um Minicomputador"
Dissertação de Mestrado - EPUSP, 1972.

2. SOUZA, BENÍCIO JOSÉ DE
"Software de um Minicomputador"
Dissertação de Mestrado a ser publicada.

3. LEE, JOHN A.N.
"The Anatomy of a Compiler"
Van Nostrand Reinhold Company, 1968.

4. PINHEIRO, WANNER MONTEIRO
"A linguagem Assembler do Patinho Feio"
Publicação interna do Lab. de Sistemas Digitais,1974.

5. MASSOLA, ANTONIO MARCOS DE AGUIRRA
"Automação de Projeto de Sistemas Digitiais: Simulação em Nível de Portas Lógicas"
Tese de Doutorado - EPUSP, 1974

6. HEWLWTT PACKARD
"Driver Manual"
1969

7. BARRON, D.W.
"Assemblers and Loaders"
Mac. Donald/American Elsevier Computer Monographs, 2nd. Edition, 1972

8. LANGDON. JR., GLEN GEORGE e FREGNI, EDSON
"Projeto de Computadores Digitais"
Edgard Blücher, Editora da U.S.P., 1974

9. KNUTH, DONALD E.
"Fundamental Algorithms"
Addison-Wesley Publishing Co,

10. TACHIBANA, MÁRIO
"Carregador Relocável para o Computador Pato Feio"
Publicação interna do Lab.Digitais, 1974

11. GRAHAM, ROBERT M.
"Principles of Systems Programming"
John Wiley Sons, 1975

12. BEJAMIN, ARTHUR J.
"An Extensible Editor for a Small Machine with disk storage"
Communications of the ACM vol.15 nº 8 Aug., 1972

13. BURROUGHS CORPORATION
"Burroughs B6700/B7700
Command and Edit (CANDE) language"
Information Manual, 1972

14. DONOVAN, JOHN J.
"Systems Programming"
Mc.Graw-Hill, 1972